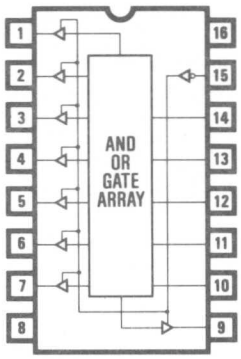
PLE™ Handbook

FIRST EDITION

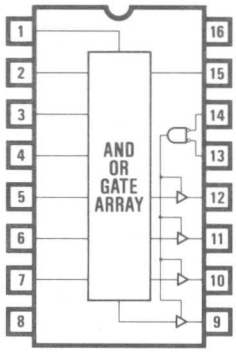
Monolithic MMI Memories

Logic Symbols

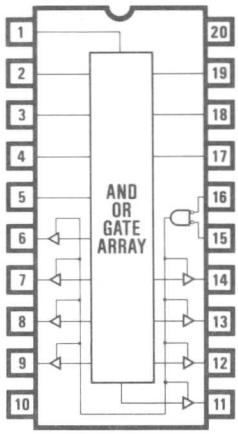
PLE5P8/A



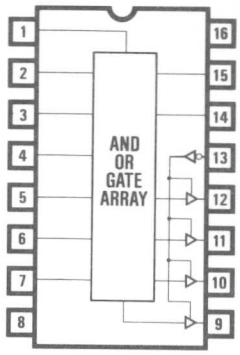
PLE8P4



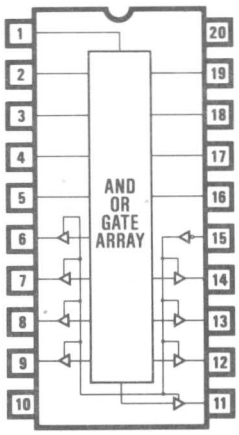
PLE8P8



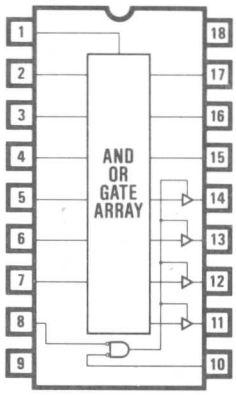
PLE9P4



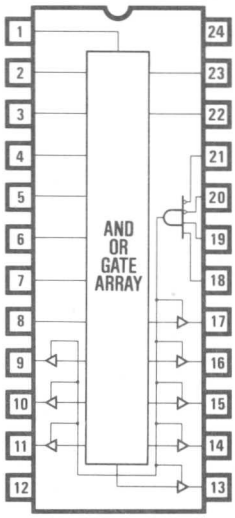
PLE9P8



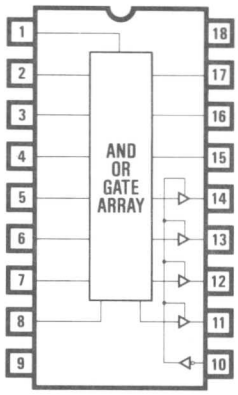
PLE10P4



PLE10P8

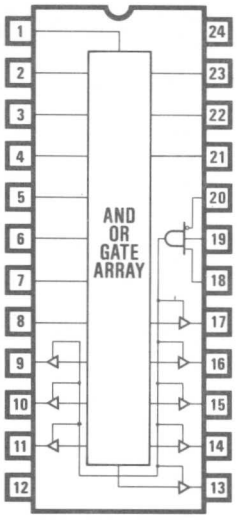


PLE11P4

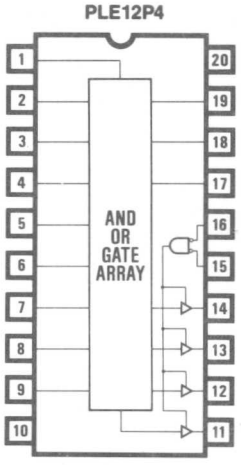


Logic Symbols

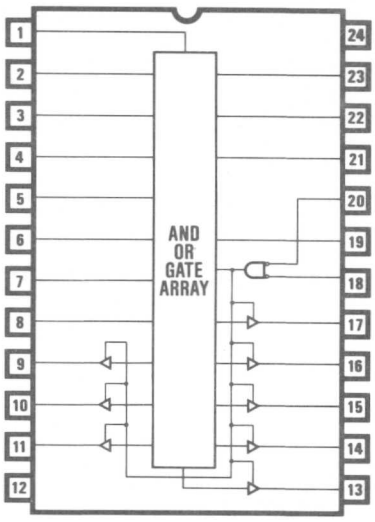
PLE11P8



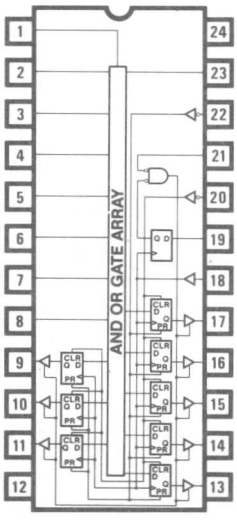
PLE12P4



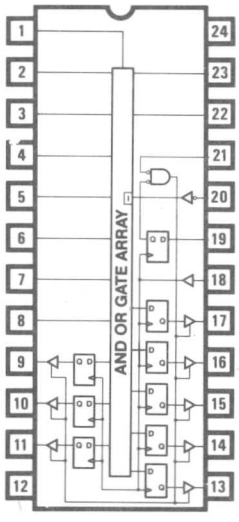
PLE12P8



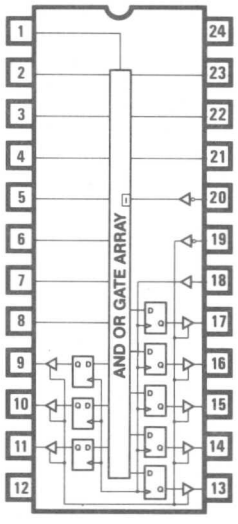
PLE9R8



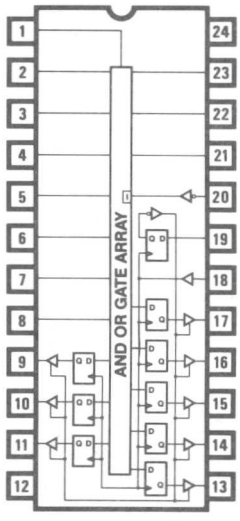
PLE10R8



PLE11RA8

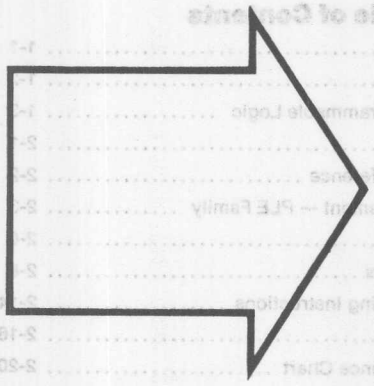


PLE11RS8



PLE™ Handbook

FIRST EDITION



Contributors: Vincent J. Coli, Zahir Ebrahim, S. Horiko, Chris Jay, C. B. Lee, Frank Lee, Ulrik Mueller, Ranjit Padmanabhan, Joel Rosenberg, Mike Vogel, Willy Voldan, Peter Wittforth, Peter Zecherle

Monolithic Memories MMI

Table of Contents

* Reprinted from EDN, February 23, 1984© CAHNERS PUBLISHING CO.

An Introduction to Programmable Logic

A logic function, whether combinatorial or sequential, may be represented in Sum of Product (SOP) form by using De Morgan's law and Boolean Algebra. Any complex multi-level logic function can easily be reduced to a two-level AND-OR configuration. This property of logic functions lends a very regular character, making it possible to implement them in a structured methodical way. The uniform AND-OR array-like architecture of Programmable Logic Devices was conceived for a clean and efficient implementation of these functions, as shown in Figure 1.

Either or both of the arrays can be programmable, constituting three distinct families of devices as shown in Figure 2.

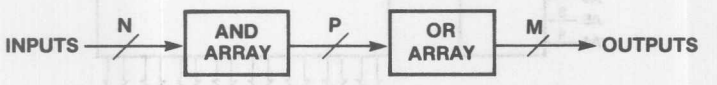


Figure 1. Structure of Programmable Logic Devices

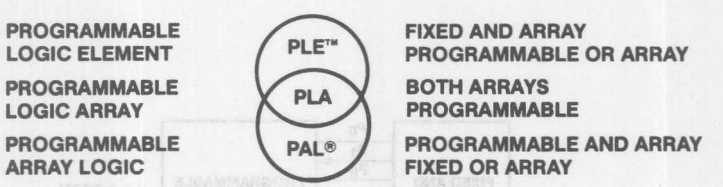


Figure 2. Structural differences between PLE, PAL and PLA

PLE Architecture

The array like structure of a PROM lends itself naturally to being viewed as a two-level AND-OR logic circuit. The inputs to the PROM are fully decoded into all possible combinations in the fixed AND plane. Each combination (product term) is fuse connected to each output in the programmable OR plane.

In terms of a PLE, a product term is equivalent to an AND gate equal in size to the number of inputs. Each output is equivalent to an OR gate connected to all the AND gates. Programming a fuse then implies breaking a connection between an AND gate and OR gate.

Thus a PROM (PLE) has a convenient structure for implementing combinatorial logic when a large number of input combinations are required, or a large number of product terms per output is desired. Registered PROMs are ideally suited for implementing complex sequential machines which contain a large number of variables in the state equations.

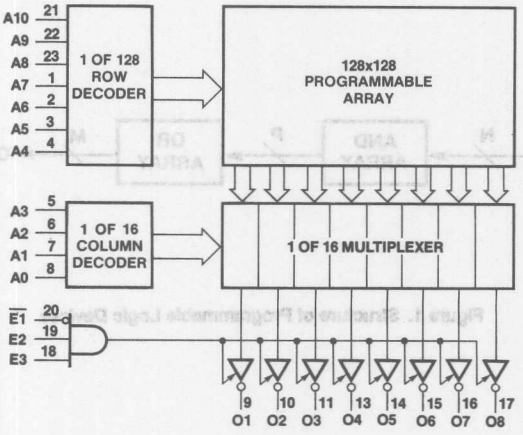


Figure 3. General Block diagram of a 2Kx8 PROM

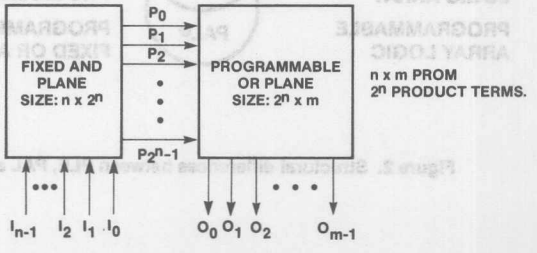


Figure 4. Block diagram of a PROM viewed as a PLE

In terms of a Karnaugh map for a PLE, each minterm in the map corresponds to one product term in the PLE. Two or more adjacent minterms cannot be combined to generate a prime implicant, or eliminate a logic hazard. For example, the following Karnaugh map implemented in a PLE will generate the function $f = a\bar{b} + ab$. The minimized function $f = a$ as indicated by the dotted prime implicant can not be implemented. The PLE does not contain a product term with fewer than all its inputs present.

The absence of prime implicants in a PLE may cause logic hazards which may be unavoidable in asynchronous control systems. However these hazards are masked out in synchronous control systems by the registers, and are largely irrelevent in data path applications where only the final steady state results are looked at. Indeed, most applications of PLEs are in synchronous control systems to replace random logic. In the data paths, PLEs are used to generate complex functions like ALU operations, high-speed multiplication, Pseudo Random Number sequences, Error Detection codes etc.

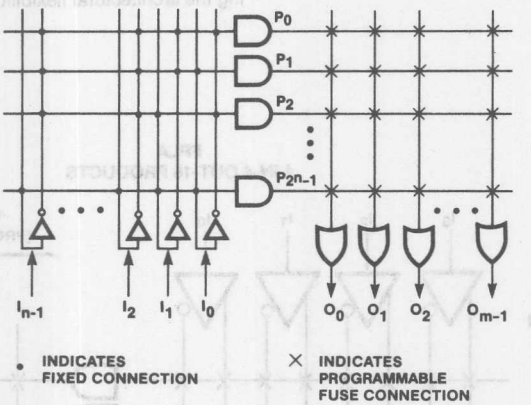


Figure 5. Equivalent Logic circuit. A virgin PLE has all the fuses intact

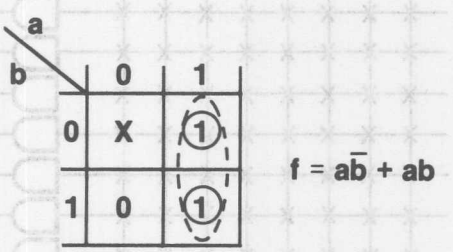


Figure 6. An arbitrary two-variable Karnaugh map for a PLE

grammable. For N inputs, M outputs, and P product terms in a PLA, the AND array contains 2 x N x P programmable crosspoint connections. All possible combinations of the inputs, taken together, or in groups, or even a single input, can have a product term in the AND array. The inputs not desired in a product term are disconnected, by removing the corresponding crosspoint connection. In field programmable PLAs, this corresponds to electrically blowing a fusible link connection. These PLAs also usually contain less product terms than the maximum possible 2^n, to conserve chip area.

tion. This architecture is used for implementing logic functions with a large number of product terms of varying sizes, or a large number of product terms per output.

Field-programmable PLAs are generally not very high performance. They are slower in speed compared to PALs and PLEs. A given signal must pass through two large programmable arrays, which increases the capacitance on the signal, and increases the delay. For most applications, a large number of crosspoints in one or another array are usually left intact, making the architectural flexibility redundant.

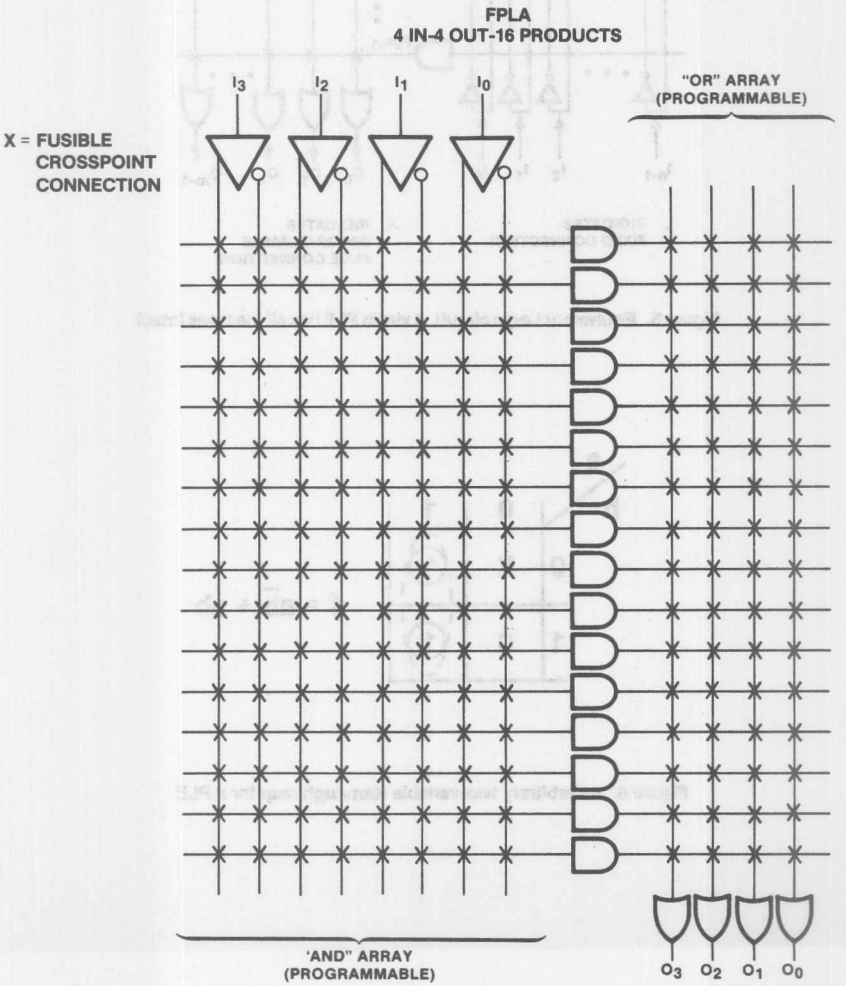


Figure 7. Structure of FPLA

PAL Architecture

PAL devices are the most useful and efficient of the fuse programmable logic family. First developed and patented by Monolithic Memories in 1976, the PALs have become well known for their friendliness in system configurations. The programmable AND array and fixed OR array eliminate most of the redundancies associated with PLAs. The PAL AND array is logically identical to a PLA AND array, with the only difference that PALs have fewer product terms. In the fixed OR array, each product term is connected to only one output or OR gate, which eliminates product term sharing of PLAs. The PAL configuration has permitted several architectural innovations, making the PAL family of devices extremely useful for implementing all kinds of logic functions. PAL features include outputs with/without registers that are internally fedback to the AND array, special XOR gates in the OR array, arithmetic carry generate gates in the feedback path in the AND array, programmable I/O pins, and programmable output polarity.

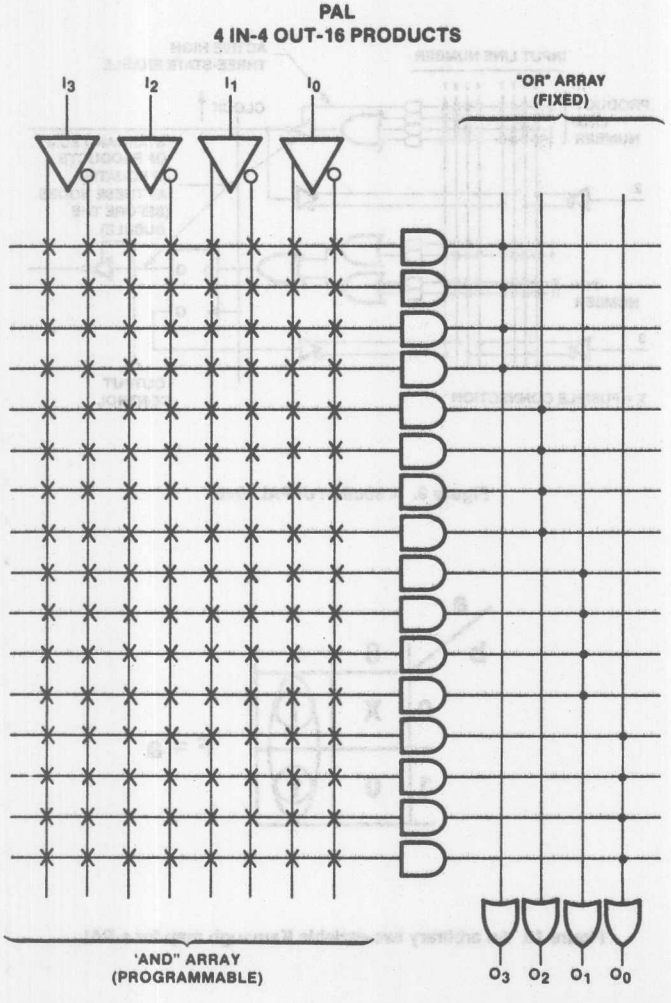


Figure 8. Structure of PAL

As the AND array in PALs is programmable, logic functions can be minimized and logic hazards removed, by combining adjacent minterms in a Karnaugh map. This group of minterms or implicants is implemented as a product term. Thus PAL outputs can be designed to be glitch-free, and ideal for implementing control logic. Figure 10 illustrates the absence of hazards and race conditions in a PAL.

The limitation of PALs is that they have a restricted number of product terms per output, and fewer product terms in general. Certain logic functions containing a large number of product terms would require a large number of PAL devices to implement them, which increases the propagation delay and the chip count. For these applications, PLEs are ideal.

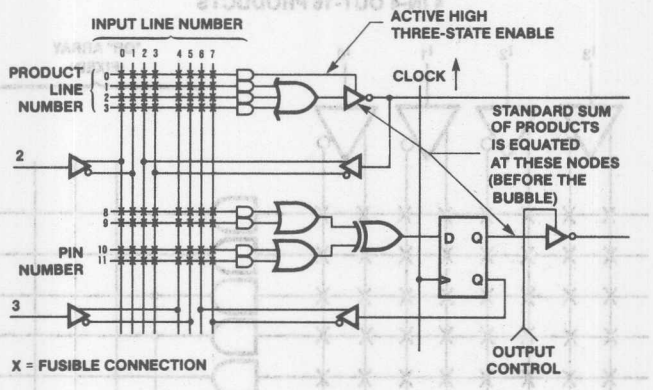


Figure 9. A section of PAL20x8

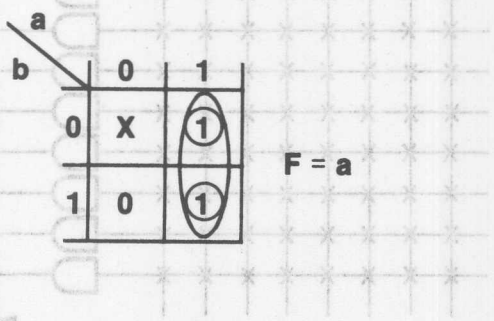


Figure 10. An arbitrary two-variable Karnaugh map for a PAL

PLEs and PALs

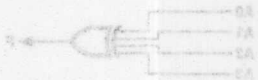
PLEs bridge the gap between the flexibility of PLAs, and the product term restrictions in PALs. Those applications for which the PALs are not suitable, the PLEs take over. Where a PAL typically has a large number of inputs, and a small number of product terms, the PLEs have a restricted number of inputs and a large number of product terms. Also, a PLE has a large number of product terms per output with full product term sharing, whereas PALs have a restricted number of product terms per output with no product term sharing. Thus PALs and PLEs complement each other both structurally and functionally.

PLE Features

- **2^n product terms per output are available with n inputs each.**
- **Programmable output polarity.**
- **Programmable initialization in registered PLEs.**
- **High-level functional specification of PLE systems in terms of logic equations.**
- **Input-to-output delays comparable to discrete logic gates (typically less than 15 ns).**
- **PNP inputs and NPN emitter follower arrays provide high-impedance and high-output current drive.**
- **Monolithic Memories' TiW fusible-link technology and advanced self-aligned washed-emitter bipolar process guarantees a programming yield greater than 98%.**

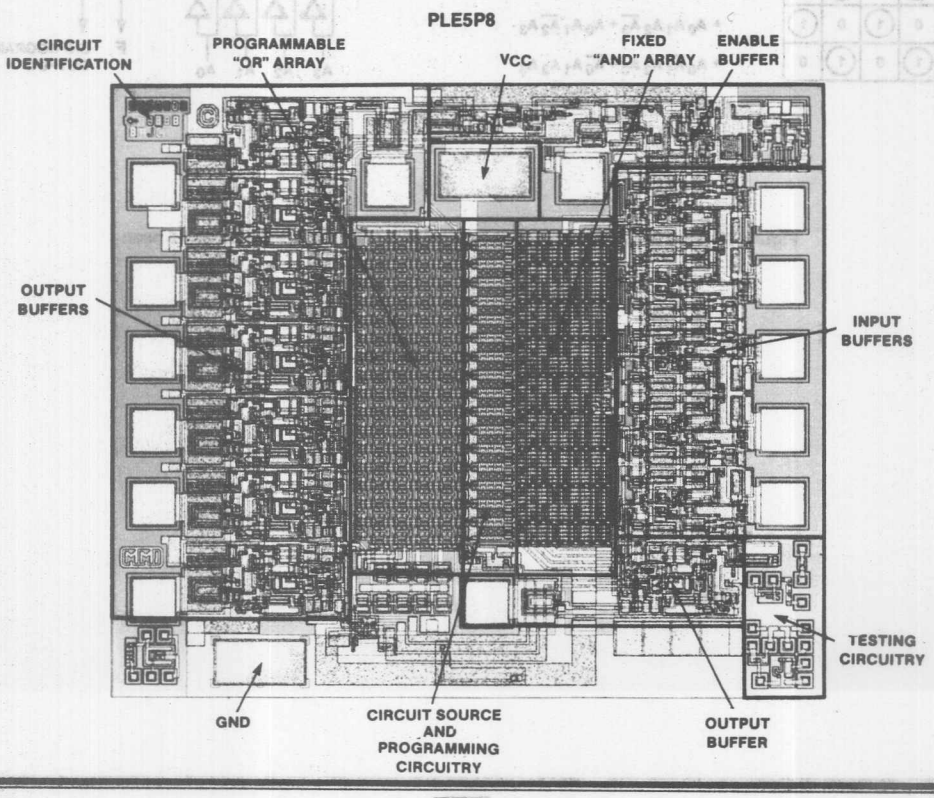
PLE Family

The PLE family of devices is an extension of Monolithic Memories' TiW PROM family. The entire line of TiW PROMs including the Registered parts are available in PLE configurations. High speed and Diagnostic versions in military and commercial temperature ranges are available.

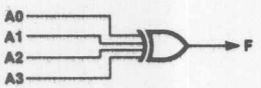


CAD Tool for PLE Designs

PLEASM is a PLE Assembler written in FORTRAN 77 and available on most mini-and microcomputer systems. It enables specifying PLE data in terms of logic equations (like those in Figure 14), which are assembled into a fuse-pattern format compatible with commercially available PROM programmers. PLEASM also generates a truth table from the logic equations which is later used as test vectors for logic simulation and verification. PLEASM allows complete design customization and documentation of PLE systems in a simple high-level functional language.



A common application of PLEs in the control path of a synchronous system, is to replace random logic, or customize logic functions. An n input exclusive OR function is quite commonly duct terms, which increases exponentially with n. Therefore, it is very efficient to implement large XOR functions in a PLE. Figure 11 shows the implementation of a 4-input XOR in a PLE.



KARNAUGH MAP

A_2, A_3 \ A_0, A_1	00	01	11	10
00	0	(1)	0	(1)
01	(1)	0	(1)	0
11	0	(1)	0	(1)
10	(1)	0	(1)	0

PLEASM EQUATIONS:

$$F = A_0 \oplus A_1 \oplus A_2 \oplus A_3$$

$$= A_0\overline{A_1}\,\overline{A_2}\,\overline{A_3} + \overline{A_0}A_1\,\overline{A_2}\,\overline{A_3}$$

$$+ \overline{A_0}\,\overline{A_1}A_2\,\overline{A_3} + \overline{A_0}\,\overline{A_1}\,\overline{A_2}A_3$$

$$+ A_0A_1A_2\overline{A_3} + A_0A_1\overline{A_2}A_3$$

$$+ A_0\overline{A_1}A_2A_3 + \overline{A_0}A_1A_2A_3$$

SECTION OF PLE5P8/A

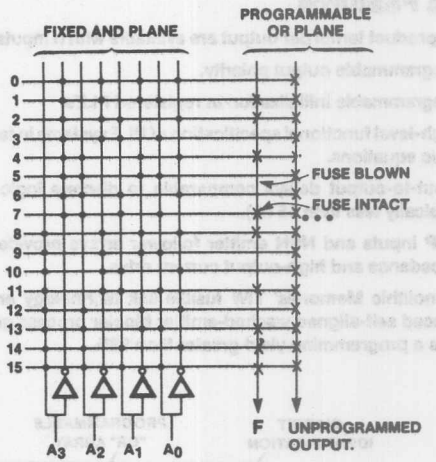


Figure 11. Four-input XOR function implemented in a PLE. MAX delay 15 nsec

Data Path Example

In the data path, a Registered PLE can be used to implement complex functions, like a Pseudo Random Number (PRN) Generator. PRN sequences are useful in encoding and decoding of information in signal processing and communication systems. They are used for data encryption in secure communication links, and error detection and correction codes in data communication systems. PRN sequences are also utilized as test vectors for circuit simulation, as signal modulators in Radar range finding systems, and as reference white noise in many signal processing applications.

Figure 12 illustrates a typical mechanism for generating PRN sequences.

The advantage of using a PLE for implementing PRN sequences is that any polynomial can be quickly customized in it. In data encryptions systems where the code is frequently changed for protection from unauthorized access, a PLE can be used to generate a new code each time, or several codes can be implemented in the same PLE. An example of a PRN generator implemented in a Registered PLE is shown in Figure 13.

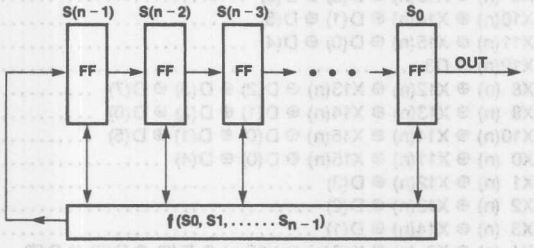


Figure 12. A 'n' Stage Linear Feedback Shift Register (LFSR)

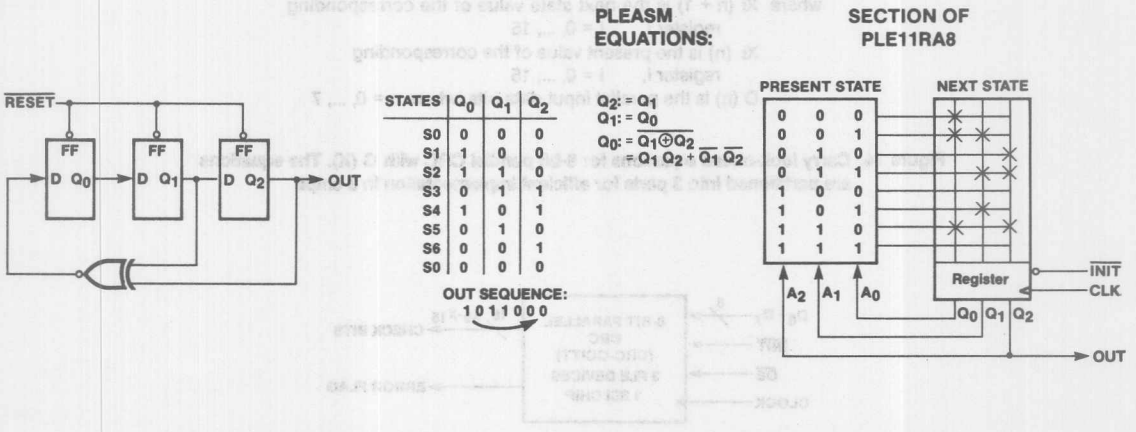


Figure 13. A 3-Stage Pseudo Random Number Generator Implemented in a Registered PLE

Parallel CRC in Registered PLEs

Implementing a high-speed M-bit Parallel Cyclic Redundancy Check (CRC) code in a Registered PLE is almost trivial. Once the M-bit carry look-ahead equations are determined, PLEASM is used to assemble these equations into a fuse pattern for the Registered PLE.

The speed of operation of parallel CRC implemented in Registered PLEs will remain the same for any generator polynomial and M. Increasing complexity of the carry look-ahead equations, only increases the number of devices required to implement them. It does not decrease the speed of operation.

To illustrate with a practical example, Figure 14 shows the 8-bit carry look-ahead equations for an 8-bit Parallel implementation of the following generator polynomial

$$G(X) = X^{16} + X^{12} + X^5 + 1$$

also called the CRC-CCITT standard. These equations are derived in Application Note AN-125, where an implementation in 4 PAL devices is also shown with a maximum delay of 90 nsec. Figure 16 shows an implementation in three 24-pin Registered PLEs and one SSI part. The maximum delay is 50 nsec.

X0 (n + 1) := X8 (n) ⊕ X12(n) ⊕ D(3) ⊕ D(7) chip1
X1 (n + 1) := X9 (n) ⊕ X13(n) ⊕ D(2) ⊕ D(6) chip2
X2 (n + 1) := X10(n) ⊕ X14(n) ⊕ D(1) ⊕ D(5) chip3
X3 (n + 1) := X11(n) ⊕ X15(n) ⊕ D(0) ⊕ D(4) chip3
X4 (n + 1) := X12(n) ⊕ D3 ..... chip1
X5 (n + 1) := X8 (n) ⊕ X12(n) ⊕ X13(n) ⊕ D(2) ⊕ D(3) ⊕ D(7) chip1
X6 (n + 1) := X9 (n) ⊕ X13(n) ⊕ X14(n) ⊕ D(1) ⊕ D(2) ⊕ D(6) chip2
X7 (n + 1) := X10(n) ⊕ X14(n) ⊕ X15(n) ⊕ D(0) ⊕ D(1) ⊕ D(5) chip3
X8 (n + 1) := X0 (n) ⊕ X11(n) ⊕ X15(n) ⊕ D(0) ⊕ D(4) chip3
X9 (n + 1) := X1 (n) ⊕ X12(n) ⊕ D(3) chip1
X10(n + 1) := X2 (n) ⊕ X13(n) ⊕ D(2) chip2
X11(n + 1) := X3 (n) ⊕ X14(n) ⊕ D(1) chip2
X12(n + 1) := X4 (n) ⊕ X8 (n) ⊕ X12(n) ⊕ X15(n) ⊕ D(0) ⊕ D(3) ⊕ D(7) chip1
X13(n + 1) := X5 (n) ⊕ X9 (n) ⊕ X13(n) ⊕ D(2) ⊕ D(6) chip2
X14(n + 1) := X6 (n) ⊕ X10(n) ⊕ X14(n) ⊕ D(1) ⊕ D(5) chip3
X15(n + 1) := X7 (n) ⊕ X11(n) ⊕ X15(n) ⊕ D(0) ⊕ D(4) chip3

where Xi (n + 1) is the next state value of the corresponding register i, i = 0, ..., 15
Xi (n) is the present value of the corresponding register i, i = 0, ..., 15
D (n) is the parallel input data bits, where n = 0, ..., 7

Figure 14. Carry look-ahead equations for 8-bit parallel CRC with G (X). The equations are partitioned into 3 parts for efficient implementation in 3 chips

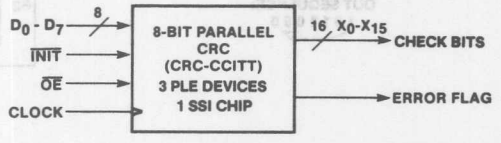


Figure 15. Block diagram of 8-bit parallel CRC

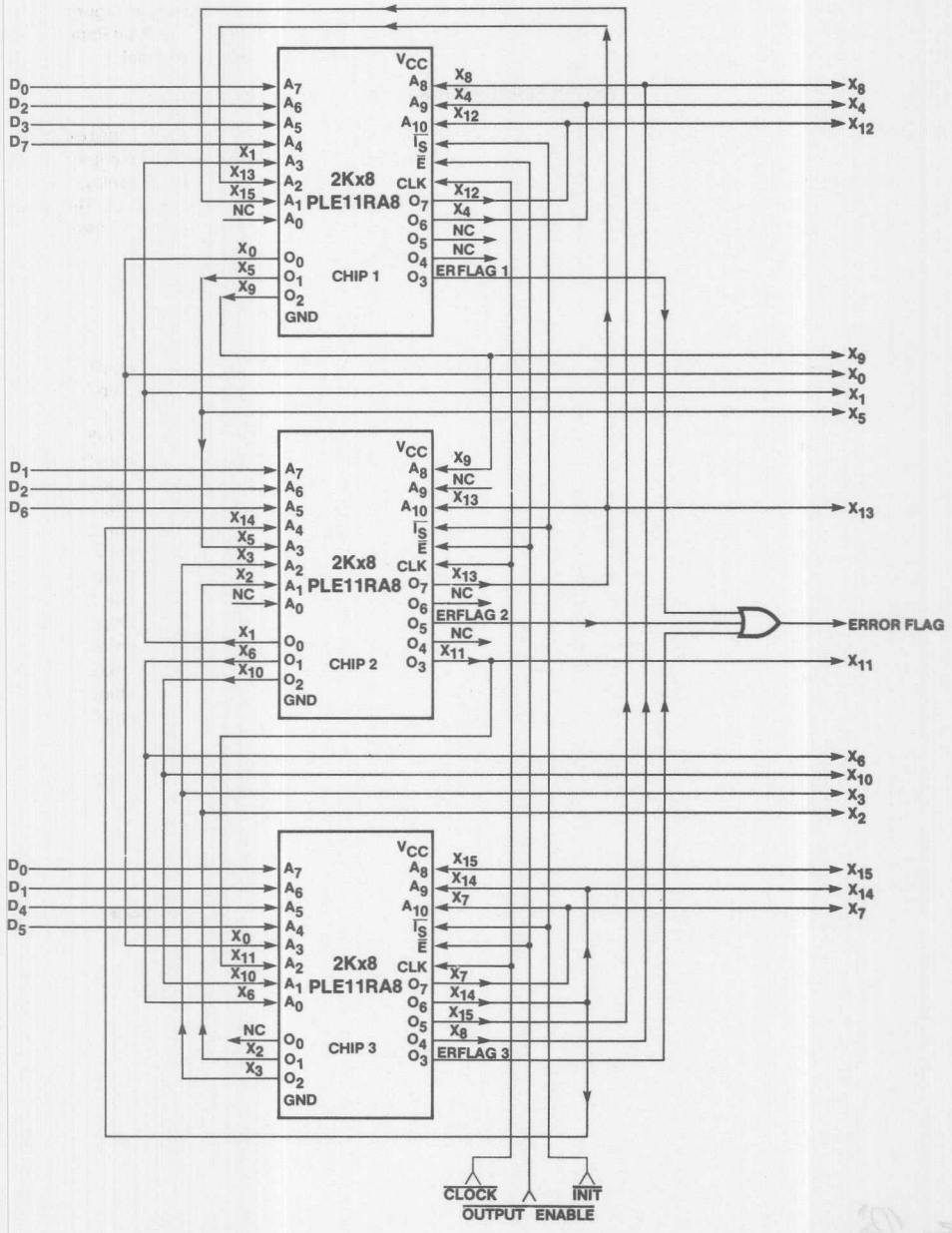
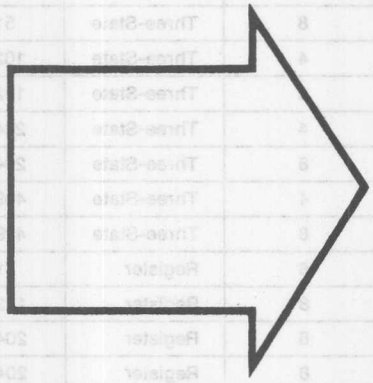


Figure 16. Diagram showing how to connect 3 Registered PLE devices to implement 8-bit parallel CRC. The Error Flag is valid on the next clock pulse after all the data has been clocked in.
Error Flag = $X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + X_8 + X_9 + X_{10} + X_{11} + X_{12} + X_{13} + X_{14} + X_{15}$


PLE
IMPORTANT.
NOTES



PLE to PROM Cross Reference

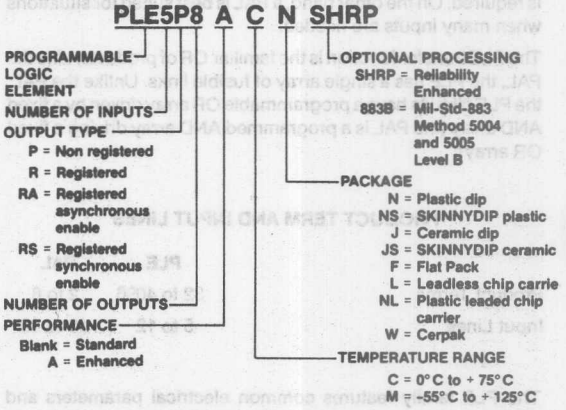
TEMP. RANGE	PLE NUMBER	INPUTS	OUTPUTS	OUTPUT TYPE	ARRAY SIZE	PROM NUMBER
Commercial	PLE5P8C	5	8	Three-State	32 x 8	63S081
	PLE5P8AC	5	8	Three-State	32 x 8	63S081A
	PLE8P4C	8	4	Three-State	256 x 4	63S141A
	PLE8P8C	8	8	Three-State	256 x 8	63S281A
	PLE9P4C	9	4	Three-State	512 x 4	63S241A
	PLE9P8C	9	8	Three-State	512 x 8	63S481A
	PLE10P4C	10	4	Three-State	1024 x 4	63S441A
	PLE10P8C	10	8	Three-State	1024 x 8	63S881A
	PLE11P4C	11	4	Three-State	2048 x 4	63S841A
	PLE11P8C	11	8	Three-State	2048 x 8	63S1681A
	PLE12P4C	12	4	Three-State	4096 x 4	63S1641A
	PLE12P8C	12	8	Three-State	4096 x 8	63S3281A
	PLE9R8C	9	8	Register	512 x 8	63RA481A
	PLE10R8C	10	8	Register	1024 x 8	63RS881A
	PLE11RA8C	11	8	Register	2048 x 8	63RA1681A
	PLE11RS8C	11	8	Register	2048 x 8	63RS1681A
Military	PLE5P8M	5	8	Three-State	32 x 8	53S081
	PLE5P8AM	5	8	Three-State	32 x 8	53S081A
	PLE8P4M	8	4	Three-State	256 x 4	53S141A
	PLE8P8M	8	8	Three-State	256 x 8	53S281A
	PLE9P4M	9	4	Three-State	512 x 4	53S241A
	PLE9P8M	9	8	Three-State	512 x 8	53S481A
	PLE10P4M	10	4	Three-State	1024 x 4	53S441A
	PLE10P8M	10	8	Three-State	1024 x 8	53S881A
	PLE11P4M	11	4	Three-State	2048 x 4	53S841A
	PLE11P8M	11	8	Three-State	2048 x 8	53S1681A
	PLE12P4M	12	4	Three-State	4096 x 4	53S1641A
	PLE12P8M	12	8	Three-State	4096 x 8	53S3281A
	PLE9R8M	9	8	Register	512 x 8	53RA481A
	PLE10R8M	10	8	Register	1024 x 8	53RS881A
	PLE11RA8M	11	8	Register	2048 x 8	53RA1681A
	PLE11RS8M	11	8	Register	2048 x 8	53RS1681A

Programmable Logic Element
PLE™ Family

Features/Benefits

- Programmable replacement for conventional TTL logic
- Reduces IC inventories and simplifies their control
- Expedites and simplifies prototyping and board layout
- Saves space with .3 inch SKINNYDIP® packages
- Programmed on standard PROM programmers
- Test and simulation made simple with PLEASM software
- Low-current PNP inputs
- Three-state outputs
- Reliable TI-W fuses guarantee >98% programming yield

Ordering Information



2

PLE Selection Guide

PART NUMBER	INPUTS	OUTPUTS	PRODUCT TERMS	OUTPUT REGISTERS	t_{PD} (ns) MAX *
PLE5P8	5	8	32		25
PLE5P8A	5	8	32		15
PLE8P4	8	4	256		30
PLE8P8	8	8	256		28
PLE9P4	9	4	512		35
PLE9P8	9	8	512		30
PLE10P4	10	4	1024		35
PLE11P4	11	4	2048		35
PLE11P8	11	8	2048		35
PLE12P4	12	4	4096		35
PLE12P8	12	8	4096		40
PLE9R8	9	8	512	8	15
PLE10R8	10	8	1024	8	15
PLE11RA8	11	8	2048	8	15
PLE11RS8	11	8	2048	8	15

* Clock to output time for registered outputs.

NOTE: Commercial limits specified.

2175 Mission College Boulevard, Santa Clara, CA 95054 Tel: (408) 970-9700 TWX: 910-338-2374
TWX: 910-338-2376

Monolithic Memories

Joining the world of IdeaLogic™ is a new generation of high-speed PROMs which the designer can use as *Programmable Logic Elements*. The combination of PLEs as logic elements with PALs can greatly enhance system speed while providing almost unlimited design freedom.

Basically, PLEs are ideal when a large number of product terms is required. On the other hand, a PAL is best suited for situations when many inputs are needed.

The PLE transfer function is the familiar OR of products. Like the PAL, the PLE has a single array of fusible links. Unlike the PAL, the PLE circuits have a programmable OR array driven by a fixed AND array (the PAL is a programmed AND array driving a fixed OR array).

PRODUCT TERM AND INPUT LINES

	PLE	PAL
Product Terms	32 to 4096	2 to 8
Input Lines	5 to 12	10 to 20

The PLE family features common electrical parameters and programming algorithm, low-current PNP inputs, full Schottky clamping and three-state outputs.

The entire PLE family is programmed on conventional PROM programmers with the appropriate personality cards and socket adapters.

output enable control through synchronous and asynchronous enable inputs, and flexible start-up sequencing through programmable initialization.

Data is transferred into the output registers on the rising edge of the clock. Provided that the asynchronous ($\overline{E}$) and synchronous ($\overline{ES}$) enables are Low, the data will appear at the outputs. Prior to the positive clock edge, register data are not affected by changes in addressing or synchronous enable inputs.

Data control is made flexible with synchronous and asynchronous enable inputs. Outputs may be set to the high-impedance state at any time by setting $\overline{E}$ to a High or if $\overline{ES}$ is High when the rising clock edge occurs. When V_{CC} power is first applied the synchronous enable flip-flop will be in the set condition causing the outputs to be in the high-impedance state.

A flexible initialization feature allows start-up and time-out sequencing with 1:16 programmable words to be loaded into the output registers. With the synchronous INITALIZE ($\overline{IS}$) pin Low, one of the 16 initialize words, addressed through pins 5, 6, 7 and 8 will be set in the output registers independent of all other input pins. The unprogrammed state of $\overline{IS}$ words are Low, presenting a CLEAR with $\overline{IS}$ pin Low. With all $\overline{IS}$ column words (A3-A0) programmed to the same pattern, the $\overline{IS}$ function will be independent of both row and column addressing and may be used as a single pin control. With all $\overline{IS}$ words programmed High a PRESET function is performed.

The PLE9R8 has asynchronous PRESET and CLEAR functions. With the chip enabled, a Low on the $\overline{PR}$ input will cause all outputs to be set to the High state. When the $\overline{CLR}$ input is set Low the output registers are reset and all outputs will be set to the Low state. The $\overline{PR}$ and $\overline{CLR}$ functions are common to all output registers and independent of all other data input states.

	AND	OR	OUTPUT OPTIONS
PLE	Fixed	Prog	TS, Registered Outputs, Fusible Polarity
FPLA	Prog	Prog	TS, OC, Fusible Polarity
FPGA	Prog	Prog	TS, OC, Fusible Polarity
FPLS	Prog	Prog	TS, Registered Feedback I/O
PAL	Prog	Fixed	TS, Registered Feedback I/O Fusible Polarity

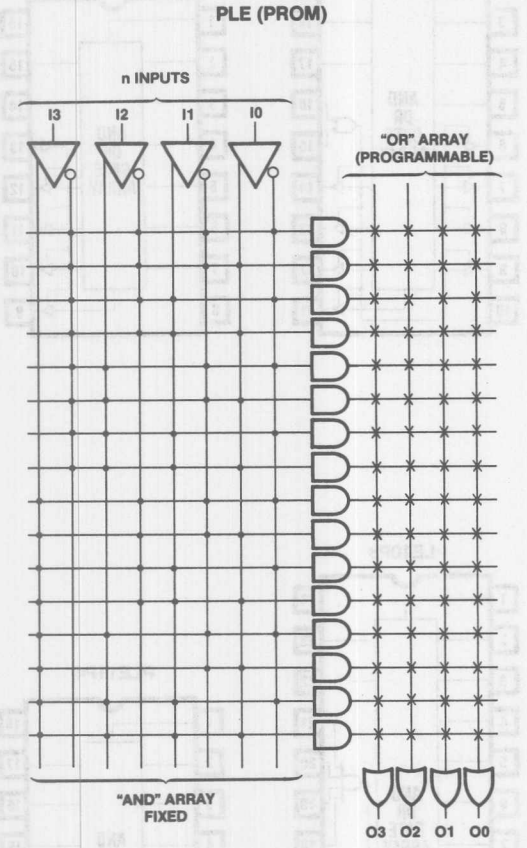
PLEASM™

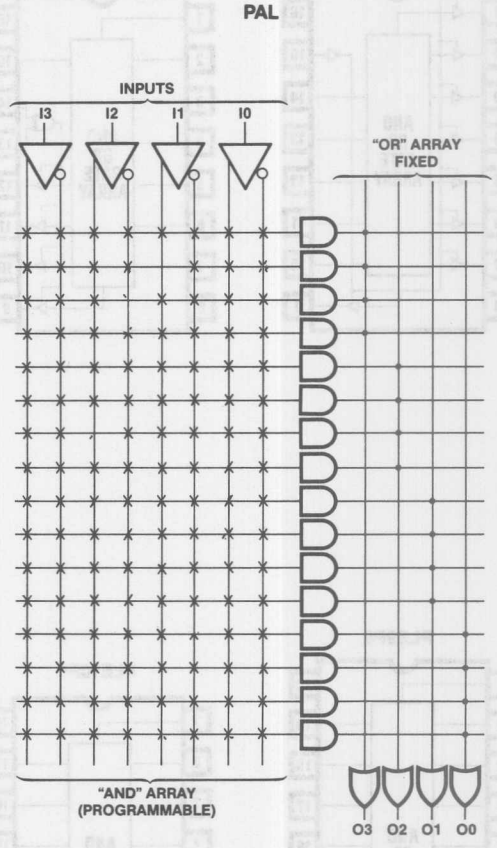
Software that makes programmable logic easy.

Monolithic Memories has developed a software tool to assist in designing and programming PROMs as PLEs. This package called "PLEASM" (PLE Assembler) is available for several computers including the VAX/VMS and IBM PC/DOS.

PLEASM converts design equations (Boolean and arithmetic) into truth tables and formats compatible with PROM programmers. A simulator is also provided to test a design using a Function Table before actually programming the PLE.

PLEASM may be requested through the Monolithic Memories IdeaLogic Exchange.





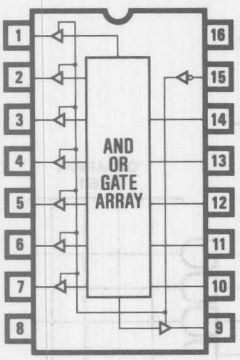
Note: • = Hardwired connection

X = Programmable fuse with a diode

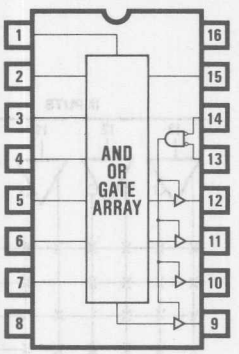
2

Logic Symbols

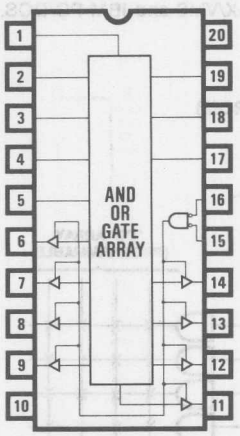
PLE5P8/A



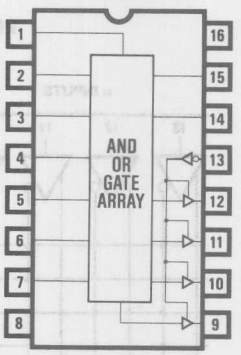
PLE8P4



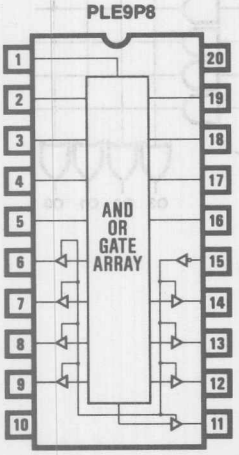
PLE8P8



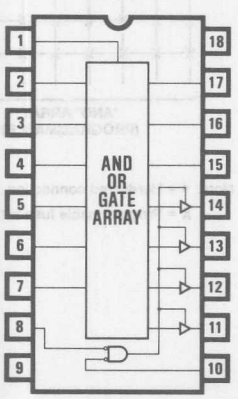
PLE9P4



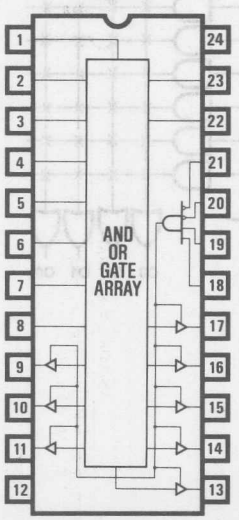
PLE9P8



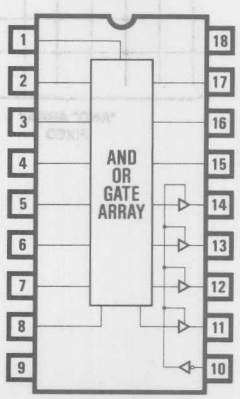
PLE10P4



PLE10P8

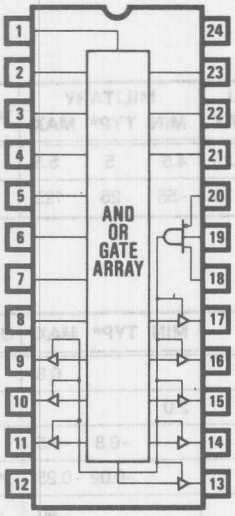


PLE11P4

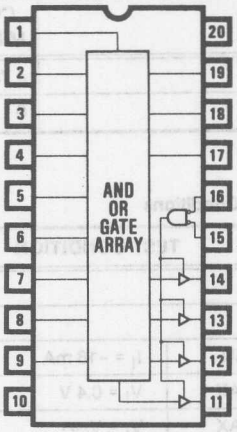


Logic Symbols

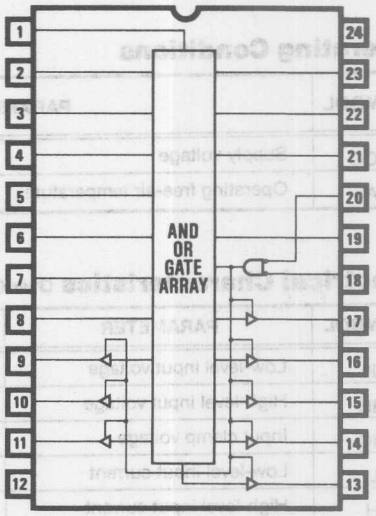
PLE11P8



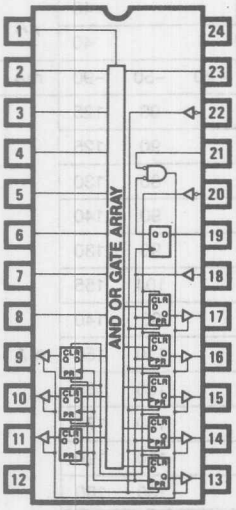
PLE12P4



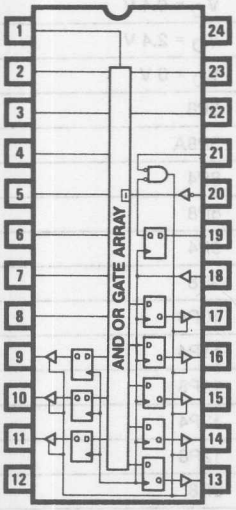
PLE12P8



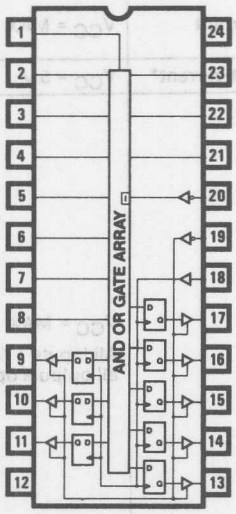
PLE9R8



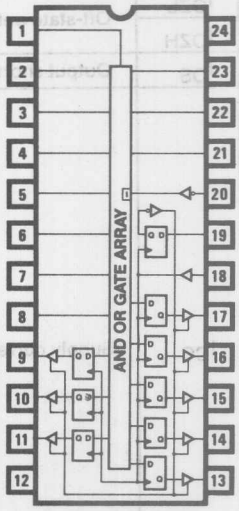
PLE10R8



PLE11RA8



PLE11RS8



2

Input voltage ..... -1.5 V to 7 V
Off-state output voltage ..... -0.5 V to 5.5 V
Storage temperature ..... -65° to +150°C

Operating Conditions

SYMBOL	PARAMETER	COMMERCIAL			MILITARY			UNIT
		MIN	TYP*	MAX	MIN	TYP*	MAX	
V_{CC}	Supply voltage	4.75	5	5.25	4.5	5	5.5	V
T_A	Operating free-air temperature	0	25	75	-55	25	125	°C

Electrical Characteristics Over Operating Conditions

SYMBOL	PARAMETER	TEST CONDITION			MIN	TYP*	MAX	UNIT
V_{IL}	Low-level input voltage						0.8	V
V_{IH}	High-level input voltage				2.0			V
V_{IC}	Input clamp voltage	V_{CC} = MIN	I_I = -18 mA			-0.8	-1.5	V
I_{IL}	Low-level input current	V_{CC} = MAX	V_I = 0.4 V			-0.02	-0.25	mA
I_{IH}	High-level input current	V_{CC} = MAX	V_I = V_{CC}				40	μA
V_{OL}	Low-level output voltage	V_{CC} = MIN	I_{OL} = 16 mA	Com		0.3	.45	V
				Mil		0.3	0.5	
V_{OH}	High-level output voltage	V_{CC} = MIN	Com I_{OH} = -3.2 mA		2.4	2.9		V
			Mil I_{OH} = -2 mA					
I_{OZL}	Off-state output current	V_{CC} = MAX	V_O = 0.4 V				-40	μA
I_{OZH}			V_O = 2.4 V				40	
I_{OS}	Output short-circuit current*	V_{CC} = 5 V	V_O = 0 V		-20	-50	-90	mA
I_{CC}	Supply current	V_{CC} = MAX All inputs TTL; all outputs open	5P8			90	125	mA
			5P8A			90	125	
			8P4			80	130	
			8P8			90	140	
			9P4			90	130	
			9P8			104	155	
			10P4			95	140	
			11P4			110	150	
			11P8			135	185	
			12P4			130	175	
			12P8			150	190	
			9R8			130	180	
			10R8			130	180	
			11RA8			140	185	
			11RS8			140	185	

* Typical at 5.0 V V_{CC} and 25° C T_A.

Switching Characteristics Over Military Operating Conditions

DEVICE TYPE	t_{PD} (ns) PROPAGATION DELAY MAX	t_{PZX} AND t_{PXZ} (ns) INPUT TO OUTPUT ENABLE/DISABLE TIME MAX
5P8AM	25	30
5P8M	35	30
8P4M	40	30
8P8M	40	30
9P4M	45	30
9P8M	40	30
10P4M	50	30
11P4M	50	30
11P8M	50	30
12P4M	50	30
12P8M	50	35

Switching Characteristics Over Commercial Operating Conditions

DEVICE TYPE	t_{PD} (ns) PROPAGATION DELAY MAX	t_{PZX} AND t_{PXZ} (ns) INPUT TO OUTPUT ENABLE/DISABLE TIME MAX
5P8AC	15	20
5P8C	25	20
8P4C	30	20
8P8C	28	25
9P4C	35	20
9P8C	30	25
10P4C	35	25
11P4C	35	25
11P8C	35	25
12P4C	35	25
12P8C	40	30

Operating Conditions

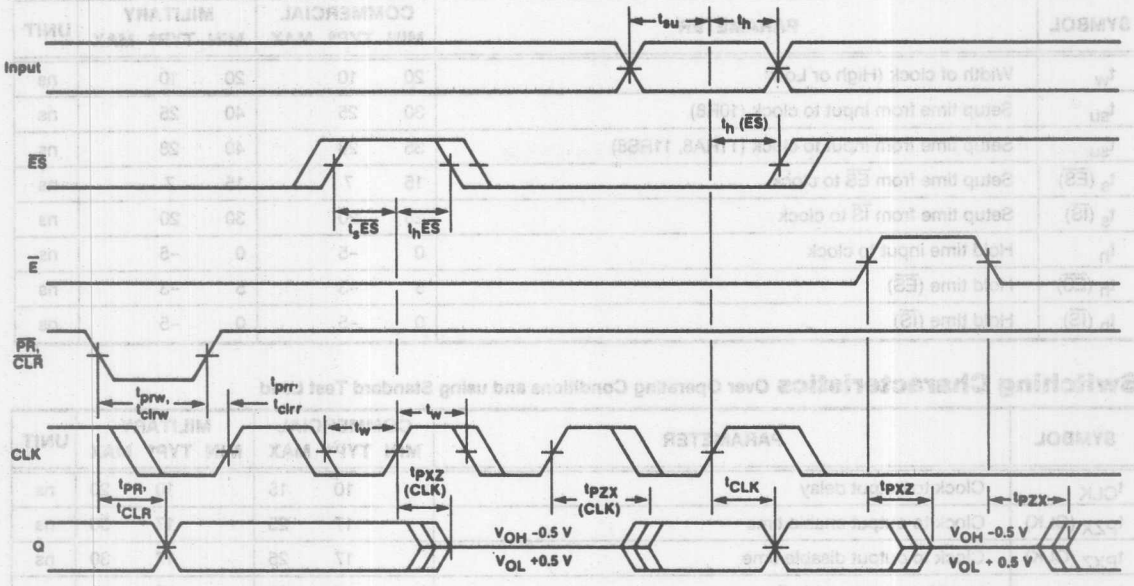
SYMBOL	PARAMETER	COMMERCIAL MIN	COMMERCIAL TYP*	COMMERCIAL MAX	MILITARY MIN	MILITARY TYP*	MILITARY MAX	UNIT
t_w	Width of clock (High or Low)	20	10		20	10		ns
t_{prw} t_{clrw}	Width of preset or clear (Low) to Output (High or Low)	20	10		20	10		ns
t_{prr} t_{clrr}	Recovery from preset or clear (Low) to clock High	20	11		25	11		ns
t_{su}	Setup time from input to clock	30	22		35	22		ns
$t_s(\overline{ES})$	Setup time from $\overline{ES}$ to clock	10	7		15	7		ns
t_h	Hold time from input to clock	0	-5		0	-5		ns
$t_h(\overline{ES})$	Hold time from $\overline{ES}$ to clock	5	-3		5	-3		ns

Switching Characteristics Over Operating Conditions and using Standard Test Load

SYMBOL	PARAMETER	COMMERCIAL MIN	COMMERCIAL TYP*	COMMERCIAL MAX	MILITARY MIN	MILITARY TYP*	MILITARY MAX	UNIT
t_{CLK}	Clock to output delay		11	15		11	20	ns
t_{PR}	Preset to output delay		15	25		15	25	ns
t_{CLR}	Clear to output delay		18	25		18	35	ns
t_{PZX} (CLK)	Clock to output enable time		14	25		14	30	ns
t_{PXZ} (CLK)	Clock to output disable time		14	25		14	30	ns
t_{PZX}	Input to output enable time		10	20		10	25	ns
t_{PXZ}	Input to output disable time		10	20		10	25	ns

* Typical at 5.0 V V_{CC} and 25°C T_A.

Definition of Waveforms



NOTES:
1. Input pulse amplitude 0 V to 3.0 V.
2. Input rise and fall times 2-5 ns from 0.8 V to 2.0 V.
3. Input access measured at the 1.5 V level.
4. Switch S_1 is closed. C_L = 30 pF and outputs measured at 1.5 V level for all tests except t_{PXZ} and t_{PZX}.
5. t_{PZX} and $t_{PZX(CLK)}$ are measured at the 1.5 V output level with C_L = 30 pF. S_1 is open for high impedance to "1" test and closed for high impedance to "0" test.

   t_{PXZ} and $t_{PXZ(CLK)}$ are tested with C_L = 5 pF. S_1 is open for "1" to high impedance test, measured at V_{OH}-0.5 V output level; S_1 is closed for "0" to high impedance test measured at V_{OL}+0.5 V output level.

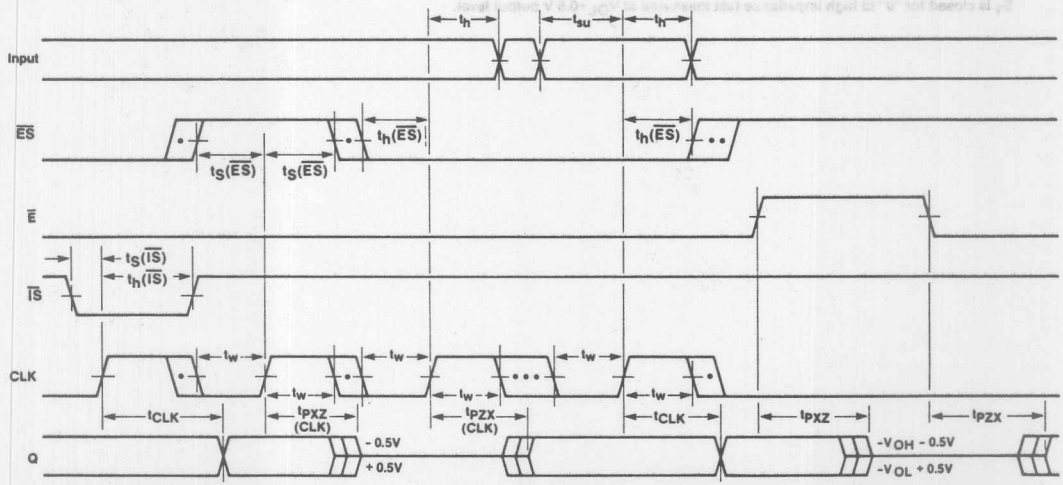
SYMBOL	PARAMETER	COMMERCIAL MIN	TYP*	MAX	MILITARY MIN	TYP*	MAX	UNIT
t_w	Width of clock (High or Low)	20	10		20	10		ns
t_{su}	Setup time from input to clock (10R8)	30	25		40	25		ns
t_{su}	Setup time from input to clock (11RA8, 11RS8)	35	28		40	28		ns
t_s ($\overline{ES}$)	Setup time from $\overline{ES}$ to clock	15	7		15	7		ns
t_s ($\overline{IS}$)	Setup time from $\overline{IS}$ to clock	25	20		30	20		ns
t_h	Hold time input to clock	0	-5		0	-5		ns
t_h ($\overline{ES}$)	Hold time ($\overline{ES}$)	5	-3		5	-3		ns
t_h ($\overline{IS}$)	Hold time ($\overline{IS}$)	0	-5		0	-5		ns

Switching Characteristics Over Operating Conditions and using Standard Test Load

SYMBOL	PARAMETER	COMMERCIAL MIN	TYP*	MAX	MILITARY MIN	TYP*	MAX	UNIT
t_{CLK}	Clock to output delay		10	15		10	20	ns
t_{PZX} (CLK)	Clock to output enable time		17	25		17	30	ns
t_{PXZ} (CLK)	Clock to output disable time		17	25		17	30	ns
t_{PZX}	Input to output enable time		17	25		17	30	ns
t_{PXZ}	Input to output disable time		17	25		17	30	ns

* Typical at 5.0 V V_{CC} and 25°C T_A.

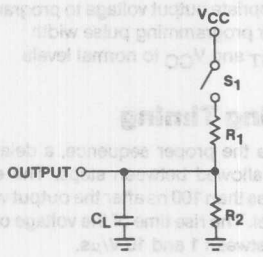
Definition of Waveforms



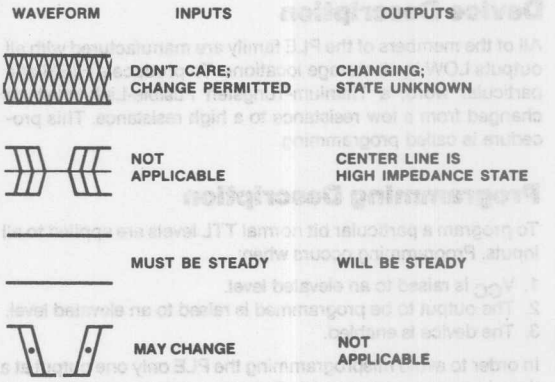
NOTES:
1. Input pulse amplitude 0 V to 3.0 V.
2. Input rise and fall times 2-5 ns from 0.8 V to 2.0 V.
3. Input access measured at the 1.5 V level.
4. Switch S_1 is closed. C_L = 30 pF and outputs measured at 1.5 V level for all tests except t_{PZX} and t_{PXZ}.
5. t_{PZX} and $t_{PZX(CLK)}$ are measured at the 1.5 V output level with C_L = 30 pF. S_1 is open for high impedance to "1" test and closed for high impedance to "0" test.

   t_{PXZ} and $t_{PXZ(CLK)}$ are tested with C_L = 5 pF. S_1 is open for "1" to high impedance test, measured at V_{OH}-0.5 V output level; S_1 is closed for "0" to high impedance test measured at V_{OL}+0.5 V output level.

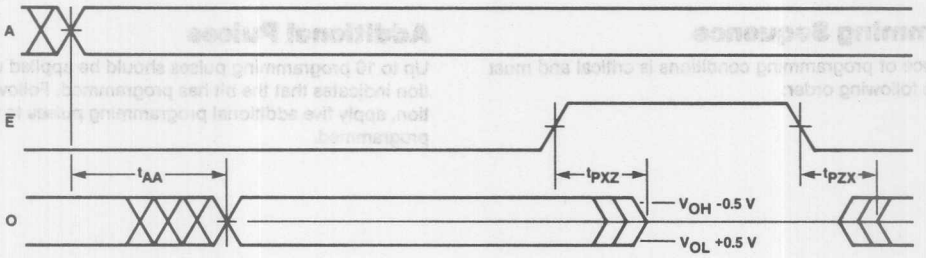
Switching Test Load



Definition of Timing Diagram



Definition of Waveforms



NOTES: Apply to electrical and switching characteristics

Typical at 5.0 V V_{CC} and 25° C T_A.

Measurements are absolute voltages with respect to the ground pin on the device and includes all overshoots due to system and/or tester noise. In all PLE devices unused inputs must be tied to either ground or V_{CC}. The series resistor required for unused inputs on standard TTL is NOT required for PLE devices, thus using less parts.

*Not more than on output should be shorted at a time and duration of the short-circuit should not exceed one second.

1. For commercial operating range R_1 = 200Ω, R_2 = 390Ω. For military operating range R_1 = 300Ω, R_2 = 600Ω.
2. Input pulse amplitude 0 V to 3.0 V.
3. Input rise and fall times 2-5 ns from 0.8 to 2.0 V.
4. Input access measured at the 1.5 V level.
5. Data delay is tested with switch S_1 closed. C_L = 30 pF and measured at 1.5 V output level.
6. t_{PZX} is measured at the 1.5 V output level with C_L = 30 pF. S_1 is open for high-impedance to "1" test and closed for high-impedance to "0" test. t_{PXZ} is measured C_L = 5 pF. S_1 is open for "1" to high-impedance test, measured at V_{OH}-0.5 V output level; S_1 is closed for "0" to high-impedance test measured at V_{OL} + 0.5 V output level.

PLE™ Family Programming Instructions

Device Description

All of the members of the PLE family are manufactured with all outputs LOW in all storage locations. To produce a HIGH at a particular word, a Titanium-Tungsten Fusible-Link must be changed from a low resistance to a high resistance. This procedure is called programming.

Programming Description

To program a particular bit normal TTL levels are applied to all inputs. Programming occurs when:

1. V_{CC} is raised to an elevated level.
2. The output to be programmed is raised to an elevated level.
3. The device is enabled.

In order to avoid misprogramming the PLE only one output at a time is to be programmed. Outputs not being programmed should be connected to V_{CC} via 5 KΩ resistors.

Unless specified, Inputs should be at VIL.

Programming Sequence

The sequence of programming conditions is critical and must occur in the following order:

1. Select the appropriate address with chip disabled
2. Increase V_{CC} to programming voltage
3. Increase appropriate output voltage to programming voltage
4. Enable chip for programming pulse width
5. Decrease V_{OUT} and V_{CC} to normal levels

Programming Timing

In order to insure the proper sequence, a delay of 100 ns or greater must be allowed between steps. The enabling pulse must not occur less than 100 ns after the output voltage reaches programming level. The rise time of the voltage on V_{CC} and the output must be between 1 and 10 V/μs.

Verification

After each programming pulse verification of the programmed bit should be made with both low and high V_{CC}. The loading of the output is not critical and any loading within the DC specifications of the part is satisfactory.

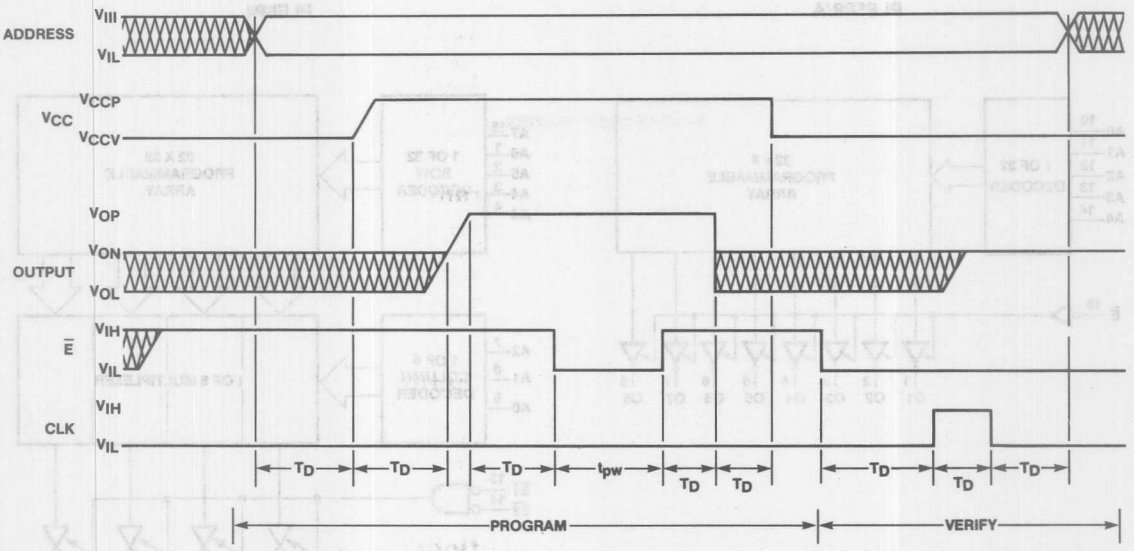
Additional Pulses

Up to 10 programming pulses should be applied until verification indicates that the bit has programmed. Following verification, apply five additional programming pulses to the bit being programmed.

Programming Parameters Do not test these parameters or you may program the device

SYMBOL	PARAMETER	RECOMMENDED MIN	RECOMMENDED VALUE	RECOMMENDED MAX	UNIT
V_{CCP}	Required V_{CC} for programming	11.5	11.75	12.0	V
V_{OP}	Required output voltage for programming	10.5	11.0	11.5	V
t_R	Rise time of V_{CC} or V_{OUT}	1.0	5.0	10.0	V/μS
I_{CCP}	Current limit of V_{CCP} supply	800	1200		mA
I_{OP}	Current limit of V_{OP} supply	15	20		mA
t_{PW}	Programming pulse width (enabled)	9	10	11	μS
V_{CC}	Low V_{CC} for verification	4.2	4.3	4.4	V
V_{CC}	High V_{CC} for verification	5.8	6.0	6.2	V
MDC	Maximum duty cycle of V_{CCP}		25	25	%
t_D	Delay time between programming steps	100	120		ns
V_{IL}	Input low level	0	0	0.5	V
V_{IH}	Input high level	2.4	3.0	5.5	V

Programming Waveforms



2

Programming Equipment Suppliers

Monolithic Memories' PLEs are designed and tested to give a programming yield greater than 98%. If your programming yield is lower, check your programmer. It may not be properly calibrated.

Programming is final manufacturing — it must be quality-controlled. Equipment must be calibrated as a regular routine, ideally under the actual conditions of use. Each time a new board or a new programming module is inserted, the whole system should be checked. Both timing and voltages must meet published specifications for the device.

Remember — The best PLEs available can be made unreliable by improper programming techniques.

SOURCE AND LOCATION

Data I/O Corp.*
10525 Willows Rd. N.E.
Redmond, WA 98052

Kontron Electronics, Inc.
630 Price Ave.
Redwood City, CA 94036

Digelec Inc.
7335 E. Acoma DR
Suite 103
Scottsdale, AZ 85260

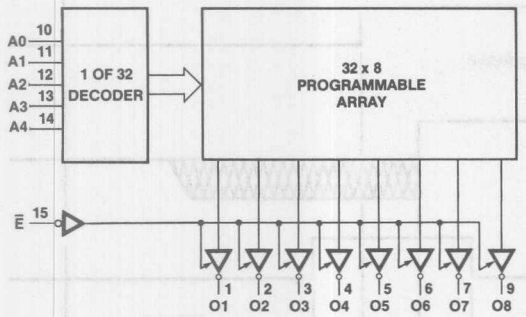
Stag Systems Inc.
1120 San Antonio Rd.
Palo Alto, CA 94303

Varix Corp.
122 Spanish Village, No. 608
Dallas, TX 75248

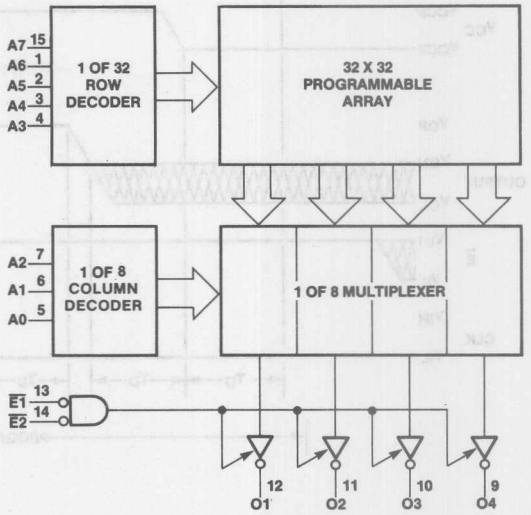
Storey Systems
3213 N. Hwy 67
Suite 103
Mesquite, TX 75150

* ABEL software also available.

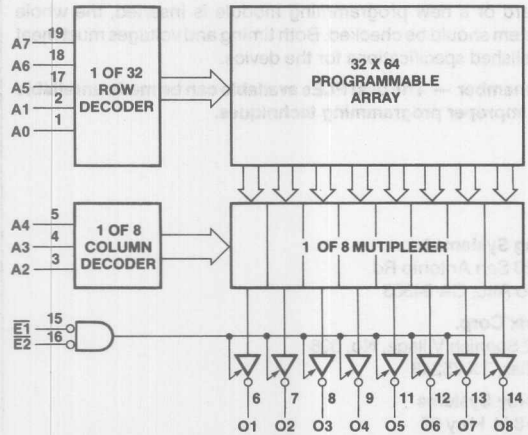
PLE5P8/A



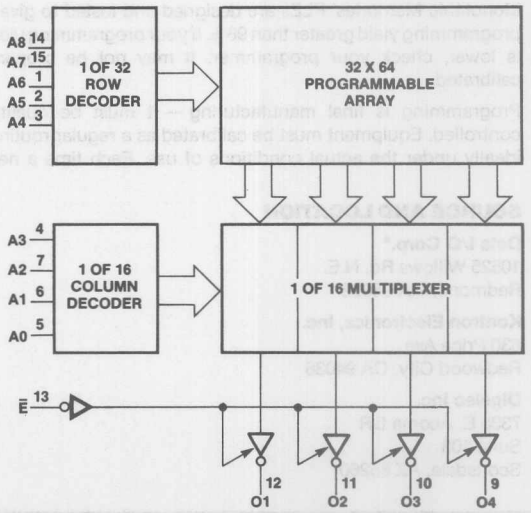
PLE8P4



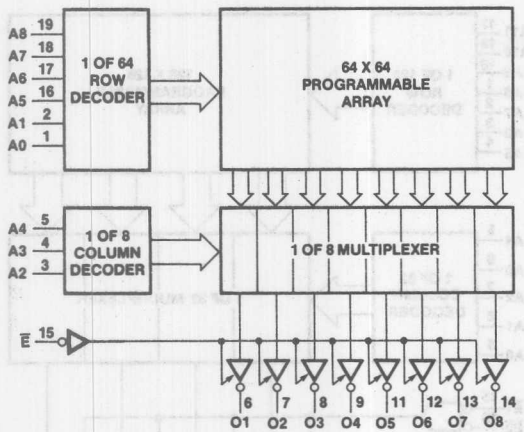
PLE8P8



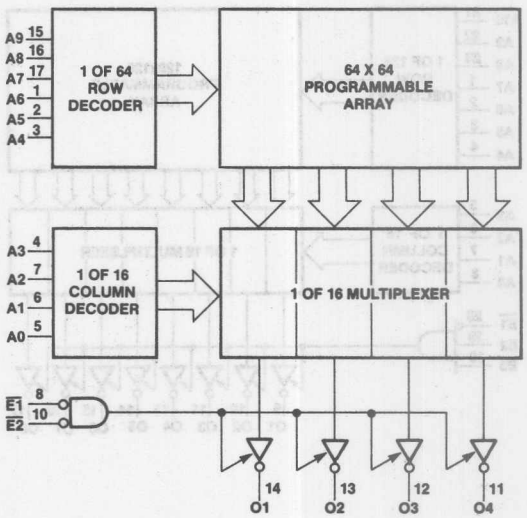
PLE9P4



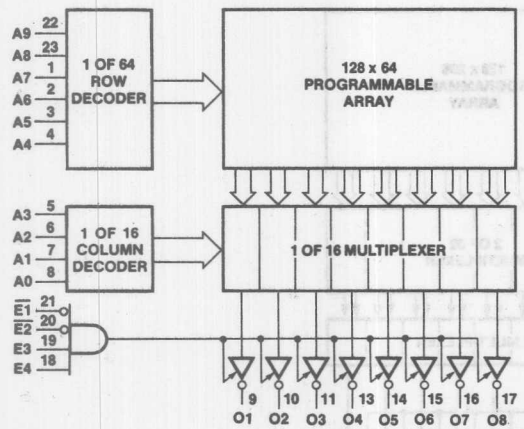
PLE9P8



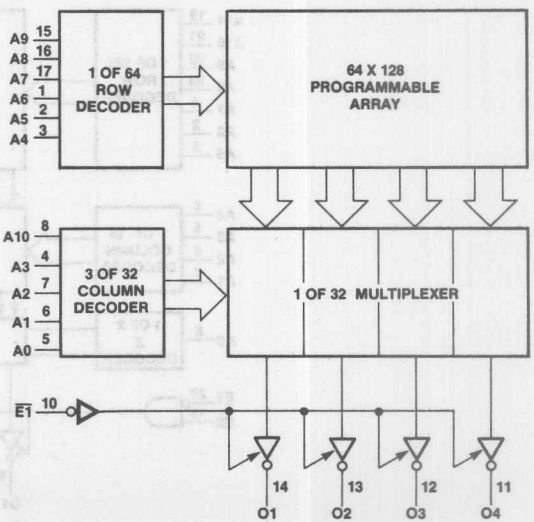
PLE10P4



PLE10P8



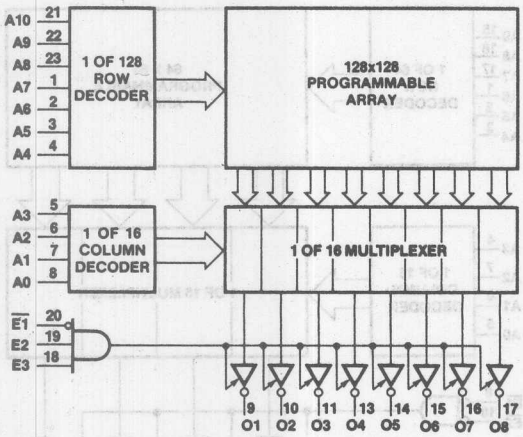
PLE11P4



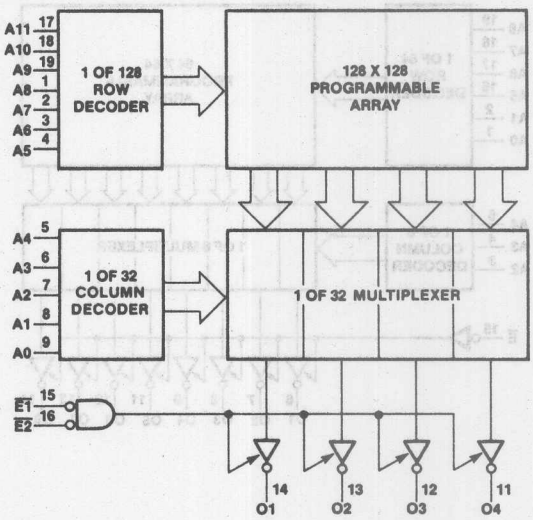
2

Block Diagrams

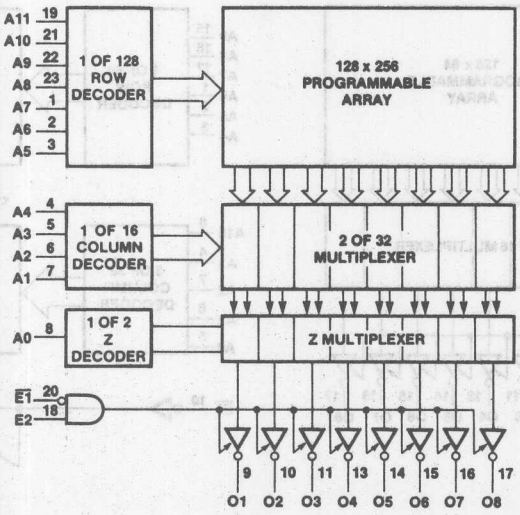
PLE11P8



PLE12P4

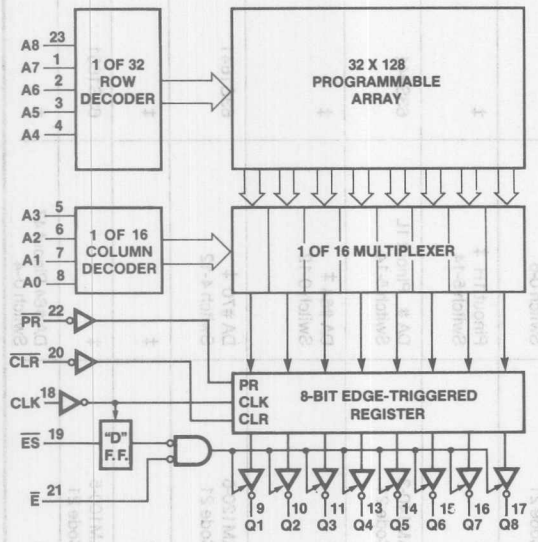


PLE12P8

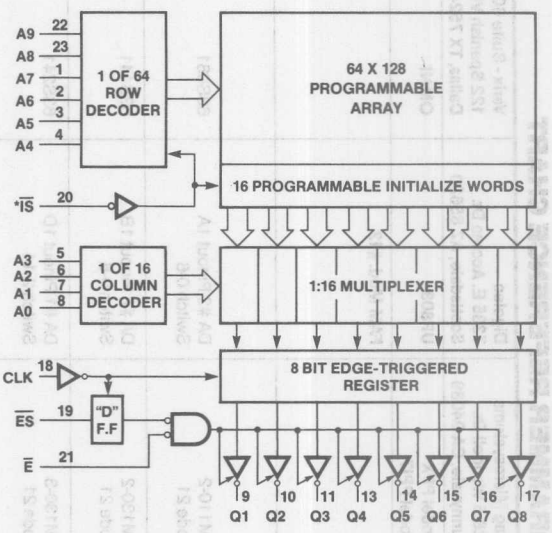


Block Diagrams

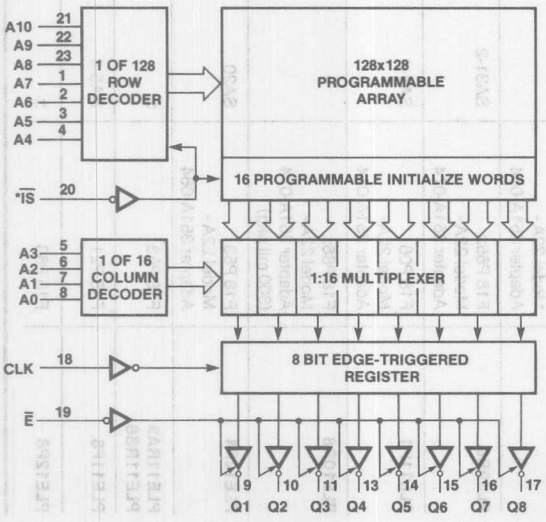
PLE9R8



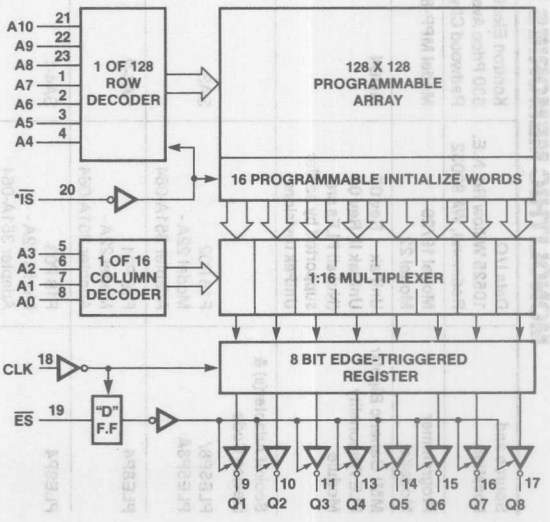
PLE10R8



PLE11RA8



PLE11RS8

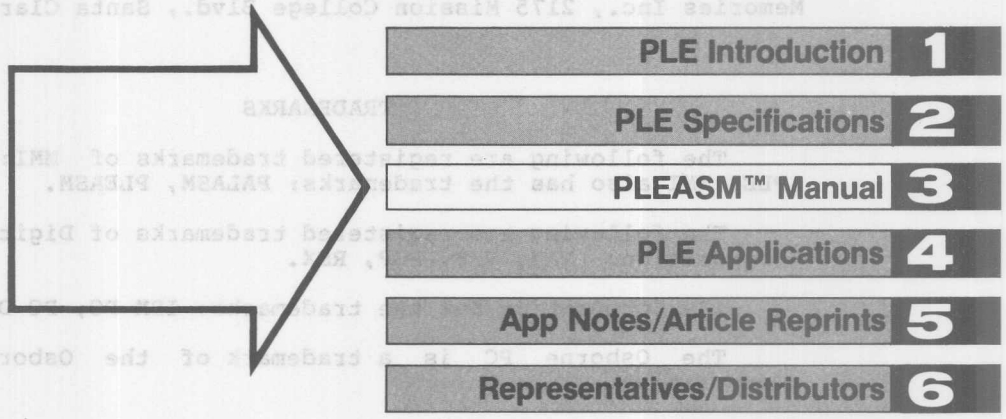


*$\overline{IS}$ selects 1:16 programmable initialization words.

MONOLITHIC MEMORIES PLE PROGRAMMER REFERENCE CHART

Source and Location	Data I/O 10525 Willow Rd. N.E. Redmond, WA 98052	Kontron Electronics 630 Price Ave. Redwood City, CA 94063	Stag Microsystems 528-5 Weddell Dr. Sunnyvale, CA 94089	Digelec 7335 E. Acoma Dr. Scottsdale, AZ 85620	Varix - Suite 608 122 Spanish Village Dallas, TX 75248
Programmer Model(s)	Model 19/29 Model 22	Model MPP-805	Model PPX Model PPI7	UP803	OMNI
MMI Generic Bipolar PLE Personality Module	UniPak Rev 07 UniPak II Rev 05 (Not all PLEs are supported by earlier UniPak revisions)	MOD4		FAM Mod. #12	
Socket Adapter(s) & Device Code					
PLE5P8/ PLE5P8A	F18 P02 Model 22A - Adapter 351A-064	SA3	AM110-2 Code 21	DA #2 Pinout 1A Switch 0-6	63S081
PLE8P4	F18 P01 Model 22A - Adapter 351A-064	SA4-2	AM130-2 Code 21	DA #2 Pinout 1B Switch 0-6	63S141
PLE9P4	F18 P03 Model 22A - Adapter 351A-064	SA4-1	AM130-3 Code 21	DA #1 Pinout 1D Switch 2-14	63S241
PLE10P4	F18 P05 Model 22A - Adapter 351A-064	SA4	AM140-2 Code 21	DA #3 Pinout 1E Switch 0-6	63S441
PLE9R8	F18 P65‡ Model 22A - Adapter 351A-074	SA31-2	‡	Pinout 1H ‡ Switch 5-14	‡
PLE11P4	F18 P06 Model 22A - Adapter 351A-064	SA4-4	AM140-3 Code 21	DA # Pinout 1L Switch 5-14	63S841
PLE10R8	F18 P86‡ Model 22A - Adapter 351A-074 (300 mil pkg)		‡	DA #64 ‡ Switch 0-12	‡
PLE12P4	F18 P53 Model 22A - Adapter 351A-064	SA20	AM120-6 Code 21	DA #70 ‡ Switch 4-12	63S1641
PLE11RA8 PLE11RS8	F18 PA3	‡	‡	‡	‡
PLE11P8	F18 P21	SA5-4	AM100-5 Code 21	‡	63S1681
PLE12P8	F18 P63	‡	‡	DA #64 Pinout 47 Switch 0-4	‡

‡ — Contact manufacturer for availability and programming information


PLE Introduction 1
PLE Specifications 2
PLEASM™ Manual 3
PLE Applications 4
App Notes/Article Reprints 5
Representatives/Distributors 6

Version 1.2

Copyright Notices:

Table of Contents

3

Introduction

PLEASM (Programmable Logic Element ASseMbler) is a software package developed by Monolithic Memories Inc. used for designing with PROM's as Programmable Logic Elements(PLE's). PLEASM is a FORTRAN IV program which assembles and simulates PLE Design Specifications. It also generates programming formats for direct download to PROM programmers and can therefore be regarded as a tool that considerably reduces the design-to-production time.

Key Features:

- Assembles Logic or Arithmetic equations into a PROM truth table.
- Provides INTEL HEX and ASCII HEX programming formats along with the hex check sum.
- Programming formats can be directly downloaded to standard PROM programmers.
- Simulates the Function Table, in the design equations.
- Reports design errors.

The purpose of this manual is to aid the user in running PLEASM and getting to know and understand all its capabilities It begins with an article that describes the wide range of PROM applications and the motivation for developing a software tool to aid in designing these applications. The next section states the system requirements for PLEASM. The next two sections give a detailed account of how to run the program with an explanation of what each of the options accomplish. This is strengthened by an example where all the operations have been performed on an input file that has the PLE specifications for basic logic gates. The PLEASM syntax is described next. This is best understood if this section is read in accompaniment with some of the design examples that come with the PLEASM program.

The Appendix gives some machine specific information about PLEASM along with details on PROM programmers, the PLE design files supplied as examples, and user customization. An important part of the Appendix is the section on the errors detected by PLEASM. This should be very useful when creating your own PLE designs.

The new PLEASM user should read Sections 3 to 5 initially and then try to run the demonstration examples. Once the user is ready to create his own design, Sections 2 and 6 will provide ideas and details of the specification format. The Appendices serve to provide additional supporting information on a number of subtleties the user should be aware of in fully utilizing the software.

PROMs vs. PLEs

PROMs have grown steadily in size and speed since their introduction in 1971, when Monolithic Memories introduced the world's first 1K bit bipolar PROM. Today, 16K and 32K PROMs are readily available and their speeds have improved such that the maximum address time (address to output) of these devices is down to 15-35 nanoseconds , over commercial operating temperature and VCC ranges. This means that PROMs can be used effectively in both high speed memory and logic replacement applications.

The PROM implements a sum-of products boolean transfer function in which any possible input (address) combination can be transferred to any output variable (data out). Figure 1.1 shows logical structure of a typical PROM. The input Fixed-AND array is a decoder and the output Programmable-OR array is a decoder. It is this decoder area that is field programmable to implement any boolean transfer function.

Each output of the AND array is connected to an input of the OR array by thin metal wire (e.g. nichrome, titanium-tungsten, platinum-silicide) which can be selectively removed from the circuit. This is referred to as "programming" or "blowing" a "fuse".

2.1 The PROM as a Memory Element

Because of their high speeds, bipolar PROMs are ideal for use in systems requiring fast Address-to Data access times. PROM applications can be found in both the data and control paths of a system.

In data paths, PROMs are used mainly as storage elements to implement different table look-up applications such as trigonometric functions, signal processing coefficients, bootstrapping and initialization programs, etc. In particular, the PROM can be used to advantage in the design of digital filters and Fast Fourier Transforms. In character generator applications, the PROM user has the flexibility of modifying the conventional fonts to his/her particular requirements.

In control paths, PROMs are used mainly to store microprograms. Microprogrammed controllers may be simple PROM-register finite state machines or they may be complex microprogrammed CPUs, where the complete instruction set of the system resides in PROM. It is thus possible, by using PROMs for microprogramming, to use the same hardware to emulate the characteristics of various processors.

Since most memory applications involve storing PROM memory data in a temporary register before it is used (pipelining), this has spawned a new generation of PROMs with on-board D-type edge-triggered registers. These registered PROMs operate faster than discrete PROM-register combinations, and the registered PROMs also occupy less space.

2. PROMs vs. PLEs

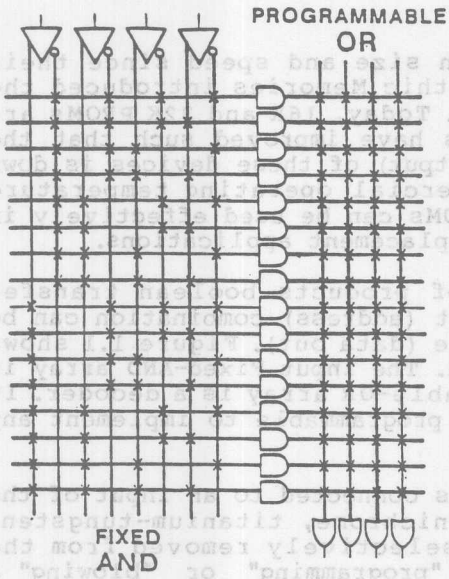


Figure 1.1 Typical PROM structure

$A \cdot B \quad C \cdot D = F_4$
$A + B + C + D = F_3$
$A \cdot B \cdot C \cdot D = F_2$
$A \cdot B = C \cdot D = F_1$

FUNCTION						F_1	F_2	F_3	F_4
ADDRESS	A_0	A_1	A_2	A_3	OUTPUT	O_1	O_2	O_3	O_4
0	0	0	0	0		1		1	
1	1	0	0	0		1		1	
2	0	1	0	0		1		1	
3	1	1	0	0		1		1	
4	0	0	1	0				1	
5	1	0	1	0				1	1
6	0	1	1	0				1	1
7	1	1	1	0		1		1	
8	0	0	0	1				1	
9	1	0	0	1				1	1
10	0	1	0	1				1	1
11	1	1	0	1		1		1	
12	0	0	1	1				1	
13	1	0	1	1				1	
14	0	1	1	1				1	
15	1	1	1	1			1	1	

Figure 1.2 Combinatorial functions available in a 16x4 PROM

2.2 The PROM as a Programmable Logic Element.

The PROM implements a sum-of-products boolean transfer function so that any function of x inputs and y outputs may be generated in a PROM with x addresses and y data outputs. Figure 1.2 shows the combinational functions available in a 16 X 4 PROM.

The AND-OR structure of the PROM can be viewed as a two-level logic circuit. The fixed AND plane contains all possible input combinations. Each input combination is a product term and it is connected to the output in the OR plane.

In terms of a PLE, a product term is the equivalent of an AND gate equal in size to the number of inputs. Each output is equivalent to an OR gate connected to all the AND gates. Programming a fuse blows this connection between the AND gate and the OR gate. The PROM thus conveniently implements combinatorial logic when a large number of input combinations are required or a large number of product terms per output is desired.

Most applications of PLEs are in synchronous control systems where they replace random logic or customise logic functions. In data paths, they are used to generate complex functions such as pseudo random number generators, ALU operations, multiplications, reciprocals, etc.

3.1 Equipment/Set-Up for Load/Go system

PLEASM should run with minimal modifications on the following CPU's provided the minimum system requirements mentioned later in this section are satisfied.

Mainframe Computers
VAX-VMS, VAX-UNIX, or IBM.

Minicomputers
PDP-11/RSX or PDP-11/RT-11

Microcomputers
IBM-PC/MS-DOS or the CP-M system on Radio Shack, Apple, Kaypro or Osborne computers.

Other requirements are:

Removable Media : 5.25'' or 8'' disks, or tape.

Memory : 64K bytes minimum.

One EIA RS232 serial communications port.

PROM programmers suggested :
DATA I/O Model 19 Programmer with Unipak.
DATA I/O Model 29 A with Unipak.
DIGELEC Model UP-803 with FAM 12.
KONTRON Model MPP-80S.
STAG Model PP17 VARIX OMNI.

3.2 Equipment/Set-Up for Development system

For software development and user customization of the program, the requirements presented earlier are necessary.

In addition, a FORTRAN compiler/linker and another disk drive are necessary to create the executable version of the program. The compiler recommended is the Supersoft FORTRAN compiler which was used to develop and test the program on microcomputers at Monolithic Memories Inc.

PLEASM is compatible with both FORTRAN IV LEVEL G and FORTRAN 77 standards. Additionally, only standard FORTRAN constructs are used to ensure portability to many computer systems.

USER'S GUIDE TO PLEASM(tm) - PLE Assembler version 1.2A

To get started with PLEASM, turn the computer ON. Check your directory to make sure you have the files mentioned in Appendix A. Once this has been verified, run the program as explained below with one of the example files. Our suggestion is to start with the file P5000.TXT which contains the PLE Design Specifications for the basic logic gates. Once the capabilities of the program have been understood, you can work with any of the other examples or attempt to create your own designs.

Using PLEASM:

Type the system's execute command to run the program. PLEASM will respond....

```
MONOLITHIC MEMORIES PLEASM(tm) VERSION 1.2A
(C) COPYRIGHT 1984 MONOLITHIC MEMORIES

WHAT IS THE SOURCE FILENAME (d:filename.ext) ?:P5000.TXT
```

At this point enter the name of the file containing the specifications for the PLE being designed. If you are using this package for the first time, we suggest you try out one of the design examples that was sent along with PLEASM. PLEASM next prompts you for the name of the file you could have the output sent to, defaulting to the console....

```
OUTPUT FILENAME - PRESS <ENTER> FOR NO OUTPUT FILE ?:<CR>
```

If you press <enter>/<return> the output will be sent to the console after each operation. At this point, the input.file is read, and a count of lines and characters in the file is written out to the screen. The next prompt is for the operation you want performed and is....

```
E=ECHO INPUT  S=SIMULATE  T=TRUTH TABLE  B=BRIEF TABLE
A=HEX TABLE   I=INTEL HEX H=ASCII HEX    C=CATALOG  Q=QUIT

ENTER OPERATION CODE:C
```

You can now enter the appropriate operation code, IN UPPER CASE!!.

The various options are briefly discussed below.

E ECHO INPUT - Prints the input PLE specifications file. Useful as a ready reference while working interactively with PLEASM.

4. Running PLEASM

S SIMULATE - Exercises the logic values in the optional function table in the logic equations provided. Errors in the function table are detected along with fairly explicit diagnostic messages. An important point to note is that all "don't care" conditions are treated as low logic values.
This option can be successfully invoked only when a function table is present in the input specifications.

T TRUTH TABLE - Prints out the entire binary truth table for all the input variables in the PLE by substitutions into the Boolean equations specified. The output has a tabular format for ease of reading. The program also provides a hex checksum for the entries in the truth table at the end.

B BRIEF TABLE - Prints out the truth table only for the used input addresses in the PLE, again by substitutions into the Boolean equations. The output is tabulated as before, this time with a partial hex checksum corresponding to the possibly shorter table.

A HEX TABLE - Prints out the entire truth table as before, except the inputs and outputs are translated into hex. Also generates a tabular format with a hex checksum.

I INTEL HEX - Generates the Intel Hex format for PROM programmers for both 4- and 8-bit data downloading. The format is shown below....

```
:AABBBB00CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDD
```

```
: --> Starting colon marker
AA --> Record Length in hex
BBBB --> Record starting address in hex
C..C --> Data
DD --> Hex checksum
```

Here all data is sent in streams of 16 8-bit bytes starting at 0000H. 4-bit data is padded up with zeros in the most significant four places. The checksum is the negative of the sum of all 8-bit bytes starting at "AA" upto "DD", modulo 256. Transmission is terminated by the string ":00000001FF".

4. Running PLEASM

H ASCII HEX SPACE - Generates the ASCII Hex format for PROM programmers for 4- and 8-bit data downloading. The format is shown below....

```
A
BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB .
C
```

A --> Record start character (STX)
BB --> Data byte
C --> End of text character (ETX)

Data is sent in streams of 16 8-bit bytes separated by spaces. An execute character, the ASCII period ".", is sent at the end of each stream of data. 4-bit data is padded up with zeros in the most significant 4 places. In addition, a hex checksum is passed at the end of the transmission.

C CATALOG - Prints a one-line description of each option provided by PLEASM. Always displays to the console.

CATALOG OF OPERATION CODES:

```
MONOLITHIC MEMORIES PLEASM(tm) VERSION 1.2A

PLEASM --PLE ASSEMBLER-- PROVIDES THE FOLLOWING OPTIONS :

C  CATALOG        - PRINTS THE PLEASM CATALOG OF OPERATIONS
E  ECHO INPUT     - PRINTS THE PLE DESIGN SPECIFICATIONS
T  TRUTH TABLE    - PRINTS THE ENTIRE TRUTH TABLE
B  BRIEF TABLE    - PRINTS ONLY USED ADDRESSES IN THE TRUTH TABLE
A  HEX TABLE      - PRINTS THE TRUTH TABLE IN HEX FORM

S  SIMULATE       - EXERCISES THE FUNCTION TABLE IN THE LOGIC
                    EQUATIONS

I  INTEL HEX      - GENERATES INTEL HEX PROGRAMMING FORMAT
H  ASCII HEX      - GENERATES ASCII HEX PROGRAMMING FORMAT

Q  QUIT           - EXITS PLEASM
```

Q QUIT - Exits the PLEASM program and prompts for restarting with another input specifications file.

5. PLEASM - An Example

```
ENTER OPERATION CODE: E
(     Echoes the input PLE Design Specifications file. This will
( help verify that the input file has been read in correctly.

PLE5P8                                          PLE DESIGN SPECIFICATION
P5000                                             VINCENT COLI 10/03/82
BASIC GATES
MMI SANTA CLARA, CALIFORNIA


O1 =  I0                                        ; BUFFER
O2 = /I0                                        ; INVERTER
O3 =  I0  *  I1  *  I2  *  I3                   ; AND GATE
O4 =  I0  +  I1  +  I2  +  I3                   ; OR GATE
O5 = /I0  + /I1  + /I2  + /I3                   ; NAND GATE
O6 = /I0  * /I1  * /I2  * /I3                   ; NOR GATE
O7 =  I0 :+: I1 :+: I2 :+: I3                   ; EXCLUSIVE OR GATE
O8 =  I0 :*: I1 :*: I2 :*: I3                   ; EXCLUSIVE NOR GATE


FUNCTION TABLE

I0 I1 I2 I3 O1 O2 O3 O4 O5 O6 O7 O8

;INPUT   -  -  OUTPUTS  FROM  BASIC  GATES  -  -
;0123    BUF  INV  AND  OR  NAND  NOR  XOR  XNOR   COMMENTS
-------------------------------------------------------------------
 LLLL     L    H    L   L    H     H    L    H     ALL ZEROS
 HHHH     H    L    H   H    L     L    L    H     ALL ONES
 HLHL     H    L    L   H    H     L    L    H     ODD CHECKERBOARD
 LHLH     L    H    L   H    H     L    L    H     EVEN CHECKERBOARD
-------------------------------------------------------------------

DESCRIPTION

THIS EXAMPLE ILLUSTRATES THE USE OF PLEs TO IMPLEMENT THE BASIC GATES:
BUFFER, INVERTER, AND GATE, OR GATE, NAND GATE, NOR GATE, EXCLUSIVE OR
GATE, AND EXCLUSIVE NOR GATE.

NOTE ALSO THAT THREE-STATE OUTPUTS ARE PROVIDED WITH ONE ACTIVE LOW
OUTPUT ENABLE CONTROL (/E).
```

```
ENTER OPERATION CODE: S
(     This  option  verifies  that  the  output  entries  in  the
( Function Table are correct for the given Boolean equations  and
( input  vectors.  Any  discrepancy between the  expected  output
( value as  given in the Function Table  and the output value  as
( computed from the Boolean equations is flagged as an error.
(
(     The  following are acceptable input entries in the Function
( Table:
(
(                H - High level
(                L - Low level
(                X - Irrelevant

FUNCTION TABLE

I0 I1 I2 I3 O1 O2 O3 O4 O5 O6 O7 O8

;INPUT   -  -   OUTPUTS  FROM  BASIC  GATES  -  -
;0123    BUF   INV  AND  OR  NAND  NOR  XOR  XNOR   COMMENTS
-------------------------------------------------------------------
 LLLL     L     H    L   L    H     H    L    H     ALL ZEROS
 HHHH     H     L    H   H    L     L    L    H     ALL ONES
 HLHL     H     L    L   H    H     L    L    H     ODD CHECKERBOARD
 LHLH     L     H    L   H    H     L    L    H     EVEN CHECKERBOARD
-------------------------------------------------------------------

PASS SIMULATION


ENTER OPERATION CODE: T
(     Generates an exhaustive binary truth table for all the given
( inputs by substitution in the Boolean equations.

BASIC GATES

 ADD    A0 A1 A2 A3 A4   O1 O2 O3 O4 O5 O6 O7 O8
-----------------------------------------------
   0    L  L  L  L  L    L  H  L  L  H  H  L  H
   1    H  L  L  L  L    H  L  L  H  H  L  H  L
   2    L  H  L  L  L    L  H  L  H  H  L  H  L
 . . . . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . . . . .
  29    H  L  H  H  H    H  L  L  H  H  L  H  L
  30    L  H  H  H  H    L  H  L  H  H  L  H  L
  31    H  H  H  H  H    H  L  H  H  L  L  L  H
-----------------------------------------------

HEX CHECK SUM = 00F48
```

5. PLEASM - An Example

```
ENTER OPERATION CODE: B
(      Prints the truth table for only the used input and  output
( pins.
BASIC GATES

 ADD   A0 A1 A2 A3  O1 O2 O3 O4 O5 O6 O7 O8
-----------------------------------------------
   0   L  L  L  L   L  H  L  L  H  H  L  H
   1   H  L  L  L   H  L  L  H  H  L  H  L
   2   L  H  L  L   L  H  L  H  H  L  H  L
 . . . . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . . . . .
  13   H  L  H  H   H  L  L  H  H  L  H  L
  14   L  H  H  H   L  H  L  H  H  L  H  L
  15   H  H  H  H   H  L  H  H  L  L  L  H
-----------------------------------------------

PARTIAL HEX CHECK SUM = 007A4


ENTER OPERATION CODE: A
(      Generates  the truth table with input and  output  vectors
( translated into hex.

BASIC GATES

 ADD       HEX ADDRESS       HEX DATA
-------------------------------------
   0            000             00B2
   1            001             0059
   2            002             005A
 . . . . . . . . . . . . . . . . . .
 . . . . . . . . . . . . . . . . . .
  29            01D             0059
  30            01E             005A
  31            01F             008D
-------------------------------------

HEX CHECK SUM = 00F48


ENTER OPERATION CODE: I
(      Generates the Intel Hex programming format for  downloading
( to a PROM programmer.

:10000000B2595A995A999A595A999A599A595A8D4C
:10001000B2595A995A999A595A999A599A595A8D3C
:00000001FF
```

3

5. PLEASM - An Example

```
ENTER OPERATION CODE: H
(      Generates the ASCII Hex Space programming format for
( downloading to a PROM programmer, with the required <STX>,
(   <SOH>,  and  <ETX> control characters delimiting the
( transmission.

^B^A
B2 59 5A 99 5A 99 9A 59 5A 99 9A 59 9A 59 5A 8D .
B2 59 5A 99 5A 99 9A 59 5A 99 9A 59 9A 59 5A 8D .
^C
00F48


ENTER OPERATION CODE: Q
(      Exits PLEASM and prompts for restarting with another input
( file.

RESTART PLEASM(Y/N) ?:
```

6. The syntax of PLEASM

PLE Design Specification:

The PLE Design Specification is the input file used with PLEASM. It is also the recommended data sheet format for describing the function of a PROM, once it has acquired the unique personality of a particular fuse pattern. This format for creating an input specifications file can be best understood by studying the examples that come along with this package. The format for the PLE Design Specification is :

Line 1 PLE part number, starting in column 1, followed by the words PLE DESIGN SPECIFICATION.

Line 2 User's part number followed by the designer's name and the date, starting in column 1. This may be an identifier that defines the application and is used for reference.

Line 3 Device Application name, starting in column 1.

Line 4 User's company name and address, starting in column 1.

Lines 2-4 are without formal rules and are provided for documentation purposes.

Line 5 Address pin list, prefixed by .ADD, starting in column 1. The pin list should be ordered LSB first.

Line 6 Data pin list, prefixed by .DAT, starting in column 1. The pin list should be ordered LSB first.

The pin list is a sequence of symbolic names separated by one or more spaces on one or more lines in the order of the device pin numbers. Each symbolic name is unique (except the unused pins which may have the same name). All pins including power and ground must be named. Names may use any printable characters except the operators ';', '.', ',', '=', '*', and '+'. The prefix '/' is used to logically complement the name.

Line m EQUATIONS

The transfer function of the device is expressed in the following form :

SYMBOL = EXPRESSION

The following terms are used to construct the equations:

SYMBOL Pin Name with optional prefix '/'

EXPRESSION A sequence of SYMBOLS separated by operators.

OPERATORS (in hierarchy of evaluation)

;	Comment follows
.	Dot operator (pin list or arithmetic operator follows)
ADD	Address pins (Inputs)
DAT	Data Pins (Outputs)
,	Delimiter, separates binary bits (MSB first)
=	Equality (combinatorial)

BOOLEAN OPERATORS

/	Complement, prefix to a pin name
*	AND (PRODUCT)
+	OR (SUM)
:+:	XOR (EXCLUSIVE OR)
:*:	XNOR (EXCLUSIVE NOR)

ARITHMETIC OPERATORS

.*.	Multiply (Arithmetic multiplication)
.+.	PLUS (Arithmetic addition)

Line n FUNCTION TABLE

The function table begins with the key word FUNCTION TABLE, starting in column 1 of a line following the equation list. It is followed by a pin list which may be in a different order and polarity from the pin list in Lines 5 and 6. The pin list is followed by a line of dashes (-'s) which is in turn followed by a list of vectors, one vector per line. One state must be specified for each pin name and optionally separated by spaces.

A vector is a sequence of states listed in the same order as the pin list and followed by an optional comment. The vector list is followed by another dashed line.

An important restriction is that blank lines are not permitted in the body of the function table. To separate logically distinct parts of the function table, however, comments can be used. In other words, blank lines are permitted with a semicolon (;) in the first column. Additionally, comments can be placed in this line. Extra blank lines might result in the simulator scanning past the end of the function table and detect "-" as an error symbol, resulting in failure to pass simulation.

6. The syntax of PLEASM

A function table is optional and need be present only for simulation to be performed. The keyword FUNCTION TABLE, however, is necessary in the input file. This should start in column 1, and should follow the equation list.

Definition of Function Table States:

Symbol	Definition	Input	Output
H	HIGH LEVEL	Drive High	Test High
L	LOW LEVEL	Drive Low	Test Low
X	IRRELEVANT	Don't Care Condition	Do Not Test

3

Line o DESCRIPTION (optional)

This begins with the keyword DESCRIPTION, starting in column 1. The device operation and application are described here. All the lines following the keyword DESCRIPTION are treated as comments. Even though the lines following may be blank, the keyword DESCRIPTION is necessary in the input file for assembly.

6. The syntax of PLEASM

An informal grammar for the input specifications file is given below:

```
PLE_SPECIFICATIONS_FILE --> PLE_TYPE_LINE
                            PLE_PATTERN_LINE
                            APPLICATION_DESCRIPTION_LINE
                            COMPANY_INFORMATION_LINE
                            ADDRESS_PIN_LIST
                            DATA_PIN_LIST
                            EQUATION_LIST
                            FUNCTION TABLE
                            [FUNCTION_TABLE_PIN_LIST]
                            [FUNCTION_TABLE]
                            DESCRIPTION
                           [COMMENTS        ]

PLE_TYPE_LINE --> PLE<PLE PART NUMBER> ... PLE DESIGN SPECIFICATIONS

PLE_PATTERN_LINE --> <REFERENCE NUMBER FOR APPLICATION, AUTHOR'S NAME>

APPLICATION_DESCRIPTION_LINE --> <APPLICATION NAME>

COMPANY_INFORMATION_LINE --> <COMPANY NAME AND ADDRESS>

ADDRESS_PIN_LIST --> .ADD <PIN NAME LIST FOR INPUT PINS>

DATA_PIN_LIST --> .DAT <PIN NAME LIST FOR OUTPUT PINS>

EQUATION_LIST --> <LIST OF BOOLEAN EQUATIONS>

FUNCTION_TABLE_PIN_LIST --> <LIST OF PINS TO BE SIMULATED>

FUNCTION_TABLE --> H|L|X <ENTRIES DEFINING LOGIC VALUES FOR THE PINS>

COMMENTS --> <COMMENTS DESCRIBING APPLICATION>

where, [..] denotes an optional expression,

       <..> denotes an informal representation for the expression,

      '___' denotes a keyword,

and    ' | ' denotes the "OR" operator.
```

6. The syntax of PLEASM

Important notes on PLEASM input specifications:

1. The specifications file must contain the keywords FUNCTION TABLE and DESCRIPTION starting in column 1 for assembly to occur.

2. All responses to PLEASM prompts should be in upper case.

3. The input specifications file should preferably be entirely in upper case.

4. The number of characters in the file, the number of lines in the file, and the number of characters in each line should be within the limits set in the I/O initialization package. The current limits set are 6000 chars/file, 250 lines/file, and 80 chars/line.

A.1 PLEASM on the VAX11/VMS

The following files should be in your tape if you have the Load/Go System :

PLEASM.EXE - This is the executable file that can be invoked to assemble your input PLE specifications.

P5000.TXT through P5017.TXT - These are the example files containing some applications. They are useful for studying how the input file should be written, and can be run with PLEASM to provide an on-line demonstration of how the program works. Details on the contents of these files can be found in Appendix D.

IOINIT.FOR - User customization package for array dimensions and I/O.

If you have ordered the Development system, you should have in addition the files :

PLEASM.FOR - This file contains the FORTRAN source for the PLEASM program and can be compiled by any FORTRAN compiler you have at your disposal.

IOLIB.FOR - This file contains the I/O features that make this version of PLEASM compatible with that on the IBM-PC. This will have to be linked in with the main program during compilation.

Unloading Your Mag Tape under VAX/VMS

Helpful Hints:

- The volume is labelled PLEASM and is recorded in Files-11 format, 1600 BPI and 9-track mag tape.

- For the neophytes who wish to learn everything there is to learn about mag tapes and more, Digital Equipment Corporation has published The Magnetic Tape Users' Guide (Order No. AA-M539A-TE).

- Your tape drive goes by many different names. For example, MTA0: or MSA0:. To list all devices on your installation, type SH DEV M<cr> when you see the $ prompt.

After loading your tape on the drive, when you see the $ prompt, type the following:

```
$ ALL MTA0: TAPE

( Allocates space for TAPE on device MTA0:

$MOUNT/OVER=IDENT TAPE

(   This mounts the tape and overrides any tape labels. Operator
( privileges are sometimes required to do this.
```

```
$CREATE/DIR [.PLEASM]
$SET DEFAULT [.PLEASM]

(   Sets up the directory appropriately.

$COPY TAPE:*.*;* []*

( This copies the tape files to your directory.

$DISMOUNT TAPE

( Dismounts tape and wraps things up.
```

DISREGARD THIS MARKED SECTION IF YOU HAVE THE LOAD/GO SYSTEM !!
****

Compile and link the source program using the following sequence of commands, to create the executable version :

```
$ FORTRAN PLEASM,IOLIB
$ LINK PLEASM,IOLIB
```

****

Using PLEASM:

Create your PLE Design Specification file using one of your system's editors.
Then type the following:

```
$ RUN PLEASM
```

The program PLEASM should now run . At this point, refer to Section 4. for step-by-step instructions on how to use the program.

Dumping a File From the VAX(VMS) to the Data I/O:

Cable Connections:

The RS-232C cable that connects the VAX-11 to the Data I/O has lines 2 and 3 reversed. The only other pins that must be connected are pins 1 and 7.

Operating Procedures:

1. Turn Data I/O power off.
2. Connect the Data I/O programmer to the modem and VT100 terminal as shown in Fig. A.1-1.
3. Turn the Data I/O programmer on.
4. Press the "SELECT" (Data I/O).

Appendix A.1: PLEASM on the VAX11/VMS

5. Enter "EB"(Data I/O)
6. Press the "START"(Data I/O).
7. Type "TY FILENAME.DAT" (VT100).
8. Press "RETURN" on the VT100.
9. Disconnect VT-100 terminal from the modem and Data I/O.
10. Reconnect Data I/O to VT-100 as shown in Fig. A.1-2.
11. Press the "SELECT" (Data I/O).
12. Press the "SELECT" (Data I/O).
13. Enter "E1" (Data I/O).
14. Press "START" (Data I/O).
15. Use VT-100 keyboard in order to communicate with the Data I/O.

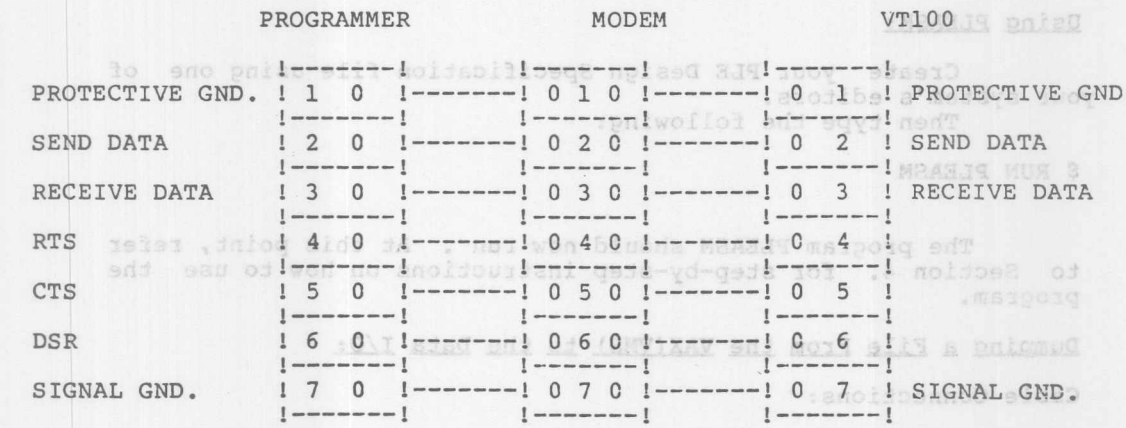


Fig. A.1-1: Downloading from host (VAX-11) to Programmer. (VAX talking to Programmer and VT100).

Appendix A.1: PLEASM on the VAX11/VMS

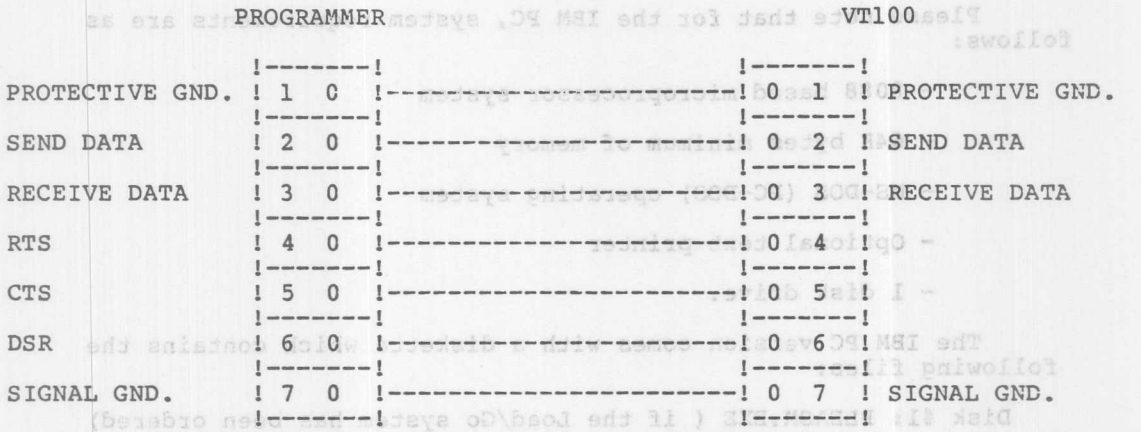


Fig. A.1-2: Using Programmer as host.

3

Appendix A.2: PLEASM on the IBM PC DOS 2.00

Please note that for the IBM PC, system requirements are as follows:

- 8088 based microprocessor system
- 64K bytes minimum of memory
- MS-DOS (PC-DOS) operating system
- Optional text printer
- 1 disk drive.

The IBM PC version comes with a diskette which contains the following files:

```
Disk #1: PLEASM.EXE ( if the Load/Go system has been ordered)
         PLEASM.FOR ( if the Development system was ordered )
         P5000.TXT - P5017.TXT
         PALCOM.EXE & PALSETUP.EXE

PLEASM.EXE is the executable version of PLEASM.
PLEASM.FOR is the FORTRAN source for the program.
P5000.TXT - P5017.TXT contain the example applications.
PALCOM.EXE is the program used for downloading.
PALSETUP.EXE allows the user to specify communications protocol

DISREGARD THIS MARKED SECTION IF YOU HAVE THE LOAD/GO SYSTEM !!
*****************************************************************

To  create the executable version for the source program you have
to   compile  and  link  the  source  file  using  any  FORTRAN
compiler/linker you have at your disposal. The one recommended is
the   Supersoft  FORTRAN  compiler  which  was  used  during  the
development  and testing of this program.  For this compiler  the
sequence of commands to create the executable file would be (with
the compiler/linker in drive B: and the source in drive A:) :

A> B:SFOR PLEASM.FOR PLEASM.REL
A> B:CNV PLEASM.REL PLEASM.OBJ/R
A> B:LINK S+SEMU+@PLEASM.RSP,PLEASM,NUL,SFLIB+MLIB,,

This creates the executable file PLEASM.EXE.

*****************************************************************
```

Appendix A.2: PLEASM on the IBM PC DOS 2.00

To use PLEASM on the IBM PC, two steps are necessary:

1. Create and edit the PLE Specification File.
2. Run PLEASM.

To run PLEASM with Disk #1 in drive A, type the following when you see the A> prompt:

```
A>PLEASM
```

At this point, PLEASM begins running. For step-by-step instructions on how to use the program, please refer to Section 4.

General:

1. Do NOT terminate the program abnormally by pressing CTRL-C, CTRL-BREAK, etc. when you are sending the output to a file rather that the screen. Use instead the QUIT option to terminate your session. If the session is terminated using CTRL-C, this may result in lost files on your disk. This is because your output file will not have been properly closed.

2. The approximate run-times for the programs vary depending on the size of the PLE and the length of the input specifications file. Simulation takes between 1 and 3 minutes, while generating programming formats could take anywhere between 3 and 10 minutes.

PROM/PLE Programmer Information:

DATA I/O Model 19 Programmer with UniPak:

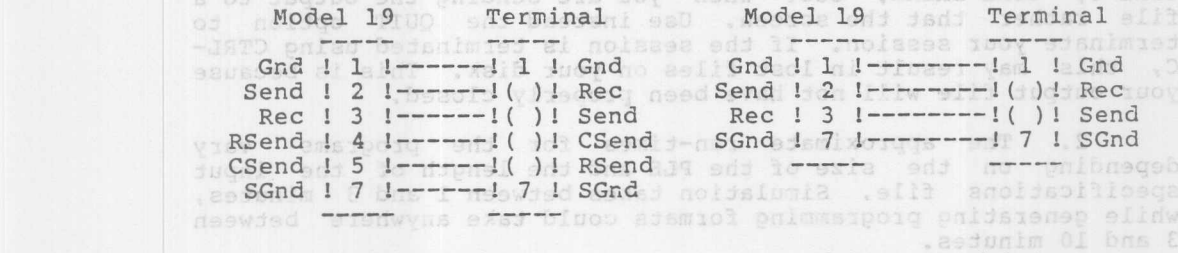
Key Features:

- Accepts PLE HEX programming format.
- Allows user limited communication via RS232 interface to computer development system.

Using the DATA I/O Model 19 with UniPak:

RS232 Serial Interface:

Prior to powering up the DATA I/O it is important that the RS232 interface be connected to the host computer correctly. The correct interface is described below:



With Handshake Without Handshake

Turn on the DATA I/O programmer. After it finishes its self check routine, type the following key sequence on the DATA I/O keyboard:

<LOAD> - Select device type

<SELECT>
<D1> - Prepares DATA I/O to receive data via RS232
<START> interface port

The DATA I/O Model 19 is now ready to accept data from the computer development system. After the file is entered, the data may be reviewed on the terminal by typing the following on the DATA I/O:

<KEYBOARD>
<ENTER> - Sends data to remote terminal

For additional information please refer to the DATA I/O Model 19 users manual.

Appendix B: PROM/PLE Programmer Information

DATA I/O Model 29A with UniPak:

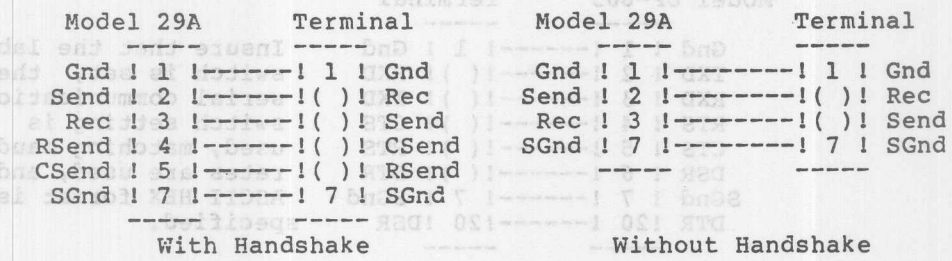
Key Features:

- Accepts PLE HEX programming format.
- Allows interactive communication via RS232 interface to computer development system.

Using the DATA I/O Model 29A with UniPak:

RS232 Serial Interface:

Prior to powering up the DATA I/O it is important that the RS232 interface be connected to the host computer correctly. The correct interface is described below:



Turn on the DATA I/O programmer. After it finishes its self check routine, type the following key sequence on the DATA I/O keyboard:

<COPY> <DEVICE> <RAM> <FFCC> - Enter family and pin code

<SELECT> <F7> - Configure for HEX input format

<COPY> <PORT> <RAM> - Prepares DATA I/O to receive data via RS232 interface port

The DATA I/O Model 29A is now ready to accept data from the computer development system. After the file is entered, the data may be reviewed on the terminal by typing the following on the DATA I/O:

<COPY> <RAM> <PORT> - Send data to remote terminal

Interactive communication may be achieved by typing on the DATA I/O:

<SELECT> <FB> - Enable output port for interactive communication

For additional information please refer to the DATA I/O Model 29A users manual.

Appendix B: PROM/PLE Programmer Information

DIGELEC Model UP-803 with FAM 12:

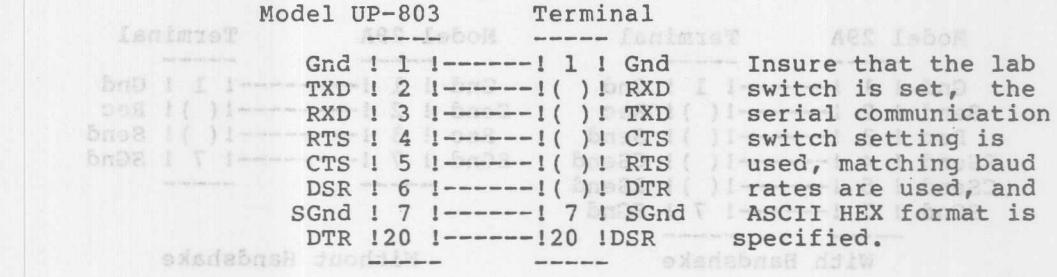
Key Features:

- Accepts PLE HEX programming format.
- Allows limited communication via RS232 interface to computer development system.

Using the DIGELEC Model UP-803 with FAM 12:

RS232 Serial Interface:

Prior to powering up the DIGELEC it is important that the RS232 interface be connected to the host computer correctly. The correct interface is described below:



Turn on the DIGELEC programmer. Prepare the DATA TRANSFER function of the UP-803 as follows:

```
SOURCE                          :   SERIAL INPUT
DESTINATION                     :   RAM
DESTINATION INITIAL ADDRESS     :   XXXX
BLOCK LENGTH                    :   YYYY ( FFFF if unknown )
Press the <EXECUTE> key on the UP-803.
```

The DIGELEC Model UP-803 is now ready to accept data from the computer development system. After the file is entered, the data may be reviewed on the remote terminal by selecting the following on the DIGELEC:

```
SOURCE                          :   RAM
DESTINATION                     :   SERIAL OUTPUT
Press the <EXECUTE> key on the UP-803.
```

For additional information on the use of the DIGELEC Model UP-803 please refer to the users manual.

Appendix B: PROM/PLE Programmer Information

KONTRON Model MPP-80S

Key Features:

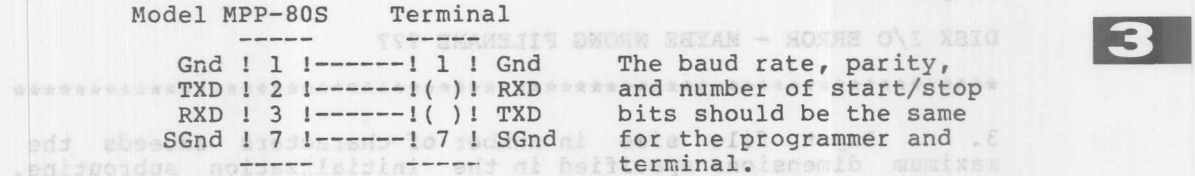
- Accepts PLE Intel HEX programming format.

- Allows limited communication via RS232 interface to computer development system.

Using the KONTRON Model MPP-80S:

RS232 Serial Interface:

Prior to powering up the KONTRON it is important that the RS232 interface be connected to the host computer correctly. The correct interface is described below:



Turn on the KONTRON programmer. Prepare the DATA TRANSFER function of the MPP-80S by performing the following:

- Select proper device type
- Press the white <IN> key (Select input port)
- Press the grey <A> key (Select port A)
- Enter <40> for Intel HEX format
- Press the grey <ENTER> key

The KONTRON Model MPP-80S is now ready to accept data from the computer development system. After the file is entered, the data may be reviewed on the remote terminal by selecting the following on the KONTRON:

- Press the white <OUT> key (Select output port)
- Press the grey <A> key (Select port A)
- Enter output format desired
- Press the grey <ENTER> key

For additional information on the use of the KONTRON model MPP-80S please refer to the users manual.

PLEASM Error messages:

PLEASM detects and reports the following errors encountered while running the program. These include incorrect file naming, syntax errors in the input specifications file, and errors in simulation.

1. Non-existent filename specified in response to the query for the name of the source file.

```
DISK I/O ERROR - MAYBE WRONG FILENAME ???
```

****

2. Open file named in response to query for the name of the output file.

```
DISK I/O ERROR - MAYBE WRONG FILENAME ???
```

****

3. Input file size in number of characters exceeds the maximum dimensions specified in the initialization subroutine. See Appendix I for details.

```
TOO MANY CHARACTERS IN INPUT FILE
```

This will probably lead to a run-time error with an output message appropriate to your system.

****

4. Keyword FUNCTION TABLE missing in input specifications.

```
*** KEYWORD "FUNCTION TABLE" MISSING. ASSEMBLY TERMINATED
```

****

5. Keyword DESCRIPTION missing in input specifications.

```
*** KEYWORD "DESCRIPTION" MISSING. ASSEMBLY TERMINATED
```

****

6. Invalid PLE name in line 1 of the input specifications.

```
PLE PART TYPE PLE$$$$ IS INCORRECT
```

****

Appendix C: PLEASM Error Messages

7. Number of pin names specified in the address list exceeds the number of input pins available in the PLE being used.

```
*** TOO MANY PIN NAMES IN INPUT PIN LIST ***

**********************************************************************
```

8. Number of pin names specified in the data list exceeds the number of output pins available in the PLE being used.

```
*** TOO MANY PIN NAMES IN OUTPUT PIN LIST ***

**********************************************************************
```

9. A pin name specified in the equations or in the function table pin list does not match any of the names declared in the address and data pin lists.

```
ERROR SYMBOL = $$$$$$$$

**********************************************************************
```

10. The "SIMULATE" option has been invoked without a function table being present in the input specifications.

```
FUNCTION TABLE MUST BE SUPPLIED IN ORDER TO PERFORM SIMULATION

**********************************************************************
```

11. The number of pin names in the function table pin list exceeds the number of pins being used in the PLE.

```
*** TOO MANY PIN NAMES IN FUNCTION TABLE PIN LIST ***

**********************************************************************
```

12. A symbol other than H(high), L(low), or X(don't care) has been entered in the function table.

```
ERROR SYMBOL **$** IN LINE $$$ OF FUNCTION TABLE

**********************************************************************
```

13. Simulation error caused by an entry in the function table (expected) not agreeing with that evaluated from the Boolean equations (actual).

```
FUNCTION TABLE ERROR IN LINE $$$ PIN = $$$$$$$$ EXPECTED $ ACTUAL $
```

The offending line in the function table is then printed out with a question mark in place of the incorrect entry.

```
**********************************************************************
```

Appendix C: PLEASM Error Messages

14. Overall count of function table errors if there are one or more simulation errors.

```
ERRORS IN FUNCTION TABLE = $$$$

**********************************************************************
```

15. PLEASM could not generate HEX programming formats for the PLE being used.

```
PLEASM DOES NOT SUPPLY HEX PROGRAMMING FORMAT FOR $$$$ BY $$ PLE'S

**********************************************************************
```

Appendix D: HELP!! And where to get it

HELP!! And where to get it:

Should you encounter any problems with PLEs or PLEASM, write or call:

The IdeaLogic Exchange
Mail-Stop (09-26)
Monolithic Memories Inc.
2175 Mission College Blvd.
Santa Clara, CA 95050

Tel: (408)970-9700

Input/Output:

Prior to running the program, the user has access to an I/O package which can be used to specify I/O unit numbers and array sizes without having to recompile the program. The variables associated with this I/O routine are explained below....

CPG(6000) - Maximum number of characters permitted in the input specifications file
LLN(250) - Maximum number of lines permitted in the input specifications file
CLN(250) - Maximum number of characters permitted per line in the input file
CONINP - The Logical Unit Number for <READ>'s from the console
CONOUT - The Logical Unit Number for <WRITE>'s to the console
FILINP - The Logical Unit Number for <READ>'s from a named file
FILOUT - The Logical Unit Number for <WRITE>'s to a named file

The Data Set Reference Numbers for INPUT and OUTPUT files are then assigned as variables at the begining of the program, as follows....

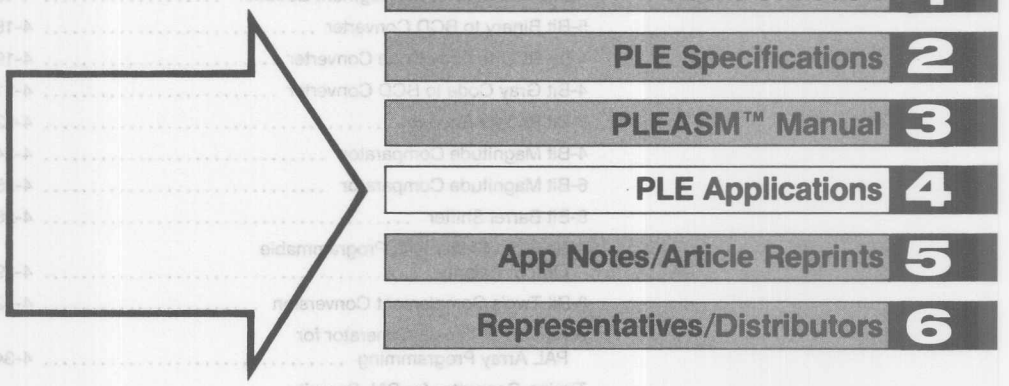
```
INPUT  PLE DESIGN SPECIFICATION                  RPD=FILINP
INPUT  OPERATION CODES                           ROC=CONINP
OUTPUT ECHO AND TRUTH TABLE                      POF=CONOUT
OUTPUT HEX AND BINARY PROGRAMMING FORMATS        PDF=CONOUT/FILOUT
OUTPUT PROMPTS AND ERROR MESSAGES                PMS=CONOUT/FILOUT
```

To perform the customization corresponding to your needs, you need to have access to the source code which means that you should have the Development system. These changes can help reduce the memory requirements and facilitate I/O.

The dimension of arrays CPG, LLN, and CLN can be modifed by making the appropriate changes in the subroutine IOINIT located at the top of the source code. This could be done to reduce memory requirements, since the arrays are statically allocated during compilation.

The logical unit numbers for the console are fixed for any given system and must be changed accordingly. For example, the logical unit number for reading from the console with VAX/VMS FORTRAN is 5 and for writing to the console is 6. These numbers are different with Supersoft FORTRAN used on the IBM-PC.

The logical unit numbers for writing and reading to and from files can be normally allocated to be between 0 and 19 excluding those reserved for the console.


PLE Introduction 1
PLE Specifications 2
PLEASM™ Manual 3
PLE Applications 4
App Notes/Article Reprints 5
Representatives/Distributors 6

Contents of Section 4

Random Logic Replacement

Random Logic Replacement

PROMs, as logic elements, have been providing solutions as replacements of random logic. This is the concept of PROM as Programmable Logic Element (PLE).

The usages of PLEs include simple multiplexer/demultiplexer/encoder/decoder, control signal generators, data communications support like CRC, and arithmetic elements like ALUs, multipliers, sine and inverse look-up tables, and applications in signal processing.

The advantages of PLEs over SSI/MSI logic devices are the flexibility of design and the fast turnaround time which non-programmable devices cannot offer. For example, if a decoder is used to select between memory pages and I/O ports, once a design is done, it will be fixed—it is not easy to find a part to be put just in the same place without modification of PC board layout in case the designer wants to expand the memory or to increase the I/O. For a PLE, what is needed is to program another PLE and place it in the same socket where the old part was placed. In addition, PLEs can allow designers to define their logic functions in a component.

The AND-OR planar structure of the PLE lends itself naturally to being viewed as a two-level logic circuit. The fixed AND plane contains all possible combinations of the literals of its inputs. Each combination (product term) is fuse-connected to each output in the programmable OR plane.

A common application of PLEs in the control path is to customize logic functions. An n input exclusive OR function is quite commonly required in comparator and adder circuits. It contains 2^{n-1} product terms, which becomes quite large for large values of n. Therefore, it is very convenient to implement large XOR functions in a PLE.

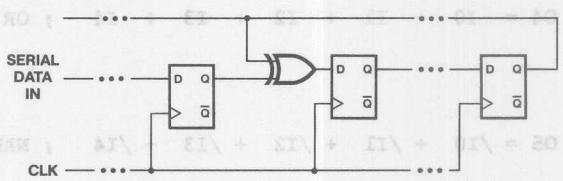
The implementation of a 4-input XOR in a PLE is shown below.

ab \ cd	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Although it seems that XOR functions may be replaced by SSIs, in most applications, the XOR functions will not be alone by themselves., PLEs can provide the flexibility of adding in additional functions without using additional packages.

In the data path, a PLE can be used to implement complex functions such as a Pseudo Random Number (PRN) Generator. Random number sequences are useful in encoding and decoding of information in signal processing and communications systems. They are used for data encryption, image quantization, waveform synchronization, and white noise generation, etc.

There are may techniques for generating PRN sequences. The most common technique, however, is to use 'n' stages of linear shift registers with feedback through a logic function. The function f is an arbitrary function chosen for a specific application. A most general linear function is an 'm' input XOR ($m \leq n$).



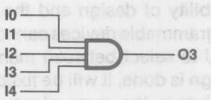
There are a number of examples in the following session which shows how a PLE can be used to replace SSI/MSI logic devices using PLEASM.

4

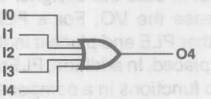
```
BASIC GATES
MMI SANTA CLARA, CALIFORNIA
.ADD I0 I1 I2 I3 I4
.DAT O1 O2 O3 O4 O5 O6 O7 O8

O1 =  I0                                  ; BUFFER
```


```
O2 = /I0                                  ; INVERTER
```

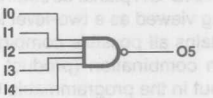

```
O3 =  I0  *  I1  *  I2  *  I3  *  I4      ; AND GATE
```



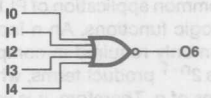

```
O4 =  I0  +  I1  +  I2  +  I3  +  I4      ; OR GATE
```



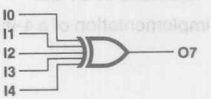

```
O5 = /I0  + /I1  + /I2  + /I3  + /I4      ; NAND GATE
```



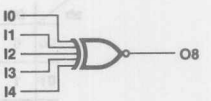

```
O6 = /I0  * /I1  * /I2  * /I3  * /I4      ; NOR GATE
```




```
O7 =  I0 :+: I1 :+: I2 :+: I3 :+: I4      ; EXCLUSIVE OR GATE
```




```
O8 =  I0 :*: I1 :*: I2 :*: I3 :*: I4      ; EXCLUSIVE NOR GATE
```



BASIC GATES (cont'd)

FUNCTION TABLE

I0 I1 I2 I3 I4 O1 O2 O3 O4 O5 O6 O7 O8

;INPUT ;01234	- - OUTPUTS FROM BASIC GATES - - BUF	INV	AND	OR	NAND	NOR	XOR	XNOR	COMMENTS
LLLLL	L	H	L	L	H	H	L	L	ALL ZEROS
HHHHH	H	L	H	H	L	L	H	H	ALL ONES
HLHLH	H	L	L	H	H	L	H	H	ODD CHECKERBOARD
LHLHL	L	H	L	H	H	L	L	L	EVEN CHECKERBOARD

DESCRIPTION

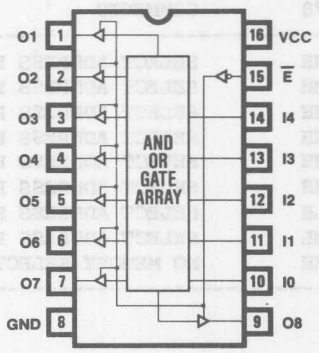
THIS EXAMPLE ILLUSTRATES THE USE OF PLEs TO IMPLEMENT THE BASIC GATES: BUFFER, INVERTER, AND GATE, OR GATE, NAND GATE, NOR GATE, EXCLUSIVE OR GATE, AND EXCLUSIVE NOR GATE.

NOTE ALSO THAT THREE-STATE OUTPUTS ARE PROVIDED WITH ONE ACTIVE LOW OUTPUT ENABLE CONTROL (/E).

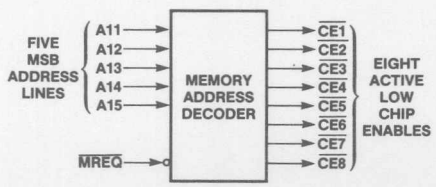
PLEASM GENERATES THE PROM TRUTH TABLE FROM THE LOGIC EQUATIONS AND SIMULATES THE FUNCTION TABLE IN THE LOGIC EQUATIONS.

BASIC GATES

PLE5P8



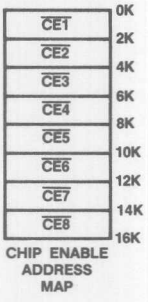
```
PLE8P8                                                PLE DESIGN SPECIFICATION
P5001                                                   ULRIK MUELLER 05/01/84
MEMORY ADDRESS DECODER
MMI SANTA CLARA, CALIFORNIA
.ADD A11 A12 A13 A14 A15 /MREQ
.DAT /CE1 /CE2 /CE3 /CE4 /CE5 /CE6 /CE7 /CE8


CE1 = /A11*/A12*/A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 0K-2K

CE2 =  A11*/A12*/A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 2K-4K

CE3 = /A11* A12*/A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 4K-6K

CE4 =  A11* A12*/A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 6K-8K

CE5 = /A11*/A12* A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 8K-10K

CE6 =  A11*/A12* A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 10K-12K

CE7 = /A11* A12* A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 12K-14K

CE8 =  A11* A12* A13*/A14*/A15* MREQ      ; SELECTS ADDRESS RANGE 14K-16K


FUNCTION TABLE

A11 A12 A13 A14 A15 /MREQ /CE1 /CE2 /CE3 /CE4 /CE5 /CE6 /CE7 /CE8

; ADD LINES
;   11111              CHIP ENABLES
;   12345      /MREQ     12345678        COMMENTS
-------------------------------------------------------------------------
    LLLLL        L       LHHHHHHH        SELECT ADDRESS RANGE 0-2K
    HLLLL        L       HLHHHHHH        SELECT ADDRESS RANGE 2K-4K
    LHLLL        L       HHLHHHHH        SELECT ADDRESS RANGE 4K-6K
    HHLLL        L       HHHLHHHH        SELECT ADDRESS RANGE 6K-8K
    LLHLL        L       HHHHLHHH        SELECT ADDRESS RANGE 8K-10K
    HLHLL        L       HHHHHLHH        SELECT ADDRESS RANGE 10K-12K
    LHHLL        L       HHHHHHLH        SELECT ADDRESS RANGE 12K-14K
    HHHLL        L       HHHHHHHL        SELECT ADDRESS RANGE 14K-16K
    XXXXX        H       HHHHHHHH        NO MEMORY SELECT (/MREQ=H)
-------------------------------------------------------------------------
```



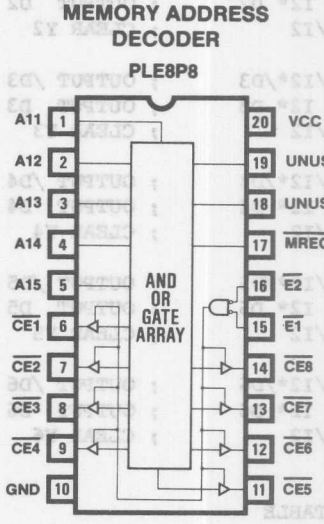
MEMORY ADDRESS DECODER (cont'd)

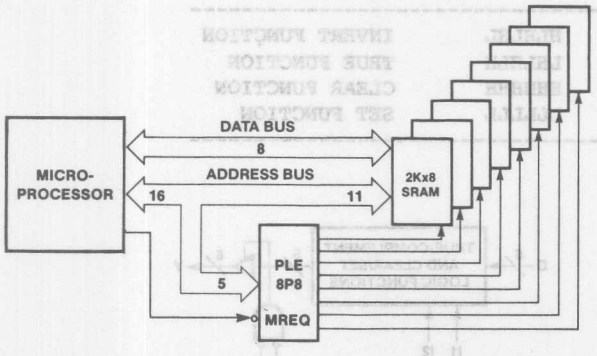
DESCRIPTION

THIS PLE8P8 PROVIDES A SINGLE CHIP ADDRESS DECODER FOR USE WITH MANY POPULAR 8-BIT MICROPROCESSORS SUCH AS THE Z80 AND 8080. THE FIVE MSB ADDRESS LINES (A11-A15) AND THE MEMORY REQUEST LINE (/MREQ) FROM THE Z80 MICROPROCESSOR ARE DECODED TO PRODUCE EIGHT ACTIVE LOW CHIP ENABLES (/CE1-/CE8) TO SELECT A RANGE OF 2K BYTES FROM A BANK OF EIGHT 2Kx8 STATIC RAMS. THIS BANK OF STATIC RAMS WILL OCCUPY THE LOWEST 16K BYTES OF ADDRESS SPACE LEAVING THE UPPER 48K BYTE SPACE AVAILABLE FOR OTHER MEMORIES AND I/O. THE PLE8P8 HAS THREE ADDITIONAL INPUTS WHICH CAN BE RESERVED FOR FUTURE SYSTEM EXPANSION.



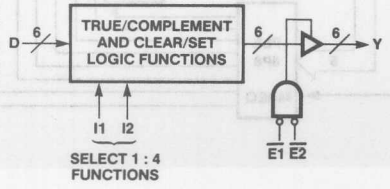
CHIP ENABLE ADDRESS MAP





4

```
6-BIT TRUE/COMPLEMENT AND CLEAR/SET LOGIC FUNCTIONS
MMI SANTA CLARA, CALIFORNIA
.ADD I1 I2 D1 D2 D3 D4 D5 D6
.DAT Y1 Y2 Y3 Y4 Y5 Y6

Y1 = /I1*/I2*/D1      ; OUTPUT /D1 (INVERT)
   + /I1* I2* D1      ; OUTPUT  D1 (TRUE)
   +  I1*/I2          ; CLEAR Y1

Y2 = /I1*/I2*/D2      ; OUTPUT /D2 (INVERT)
   + /I1* I2* D2      ; OUTPUT  D2 (TRUE)
   +  I1*/I2          ; CLEAR Y2

Y3 = /I1*/I2*/D3      ; OUTPUT /D3 (INVERT)
   + /I1* I2* D3      ; OUTPUT  D3 (TRUE)
   +  I1*/I2          ; CLEAR Y3

Y4 = /I1*/I2*/D4      ; OUTPUT /D4 (INVERT)
   + /I1* I2* D4      ; OUTPUT  D4 (TRUE)
   +  I1*/I2          ; CLEAR Y4

Y5 = /I1*/I2*/D5      ; OUTPUT /D5 (INVERT)
   + /I1* I2* D5      ; OUTPUT  D5 (TRUE)
   +  I1*/I2          ; CLEAR Y5

Y6 = /I1*/I2*/D6      ; OUTPUT /D6 (INVERT)
   + /I1* I2* D6      ; OUTPUT  D6 (TRUE)
   +  I1*/I2          ; CLEAR Y6


FUNCTION TABLE

I1 I2 D1 D2 D3 D4 D5 D6 Y1 Y2 Y3 Y4 Y5 Y6

;CONTROL     INPUT  D     OUTPUT Y
; LINES       123456       123456       COMMENTS
-------------------------------------------------------------
   LL         LHLHLH       HLHLHL       INVERT FUNCTION
   LH         LHLHLH       LHLHLH       TRUE FUNCTION
   HL         XXXXXX       HHHHHH       CLEAR FUNCTION
   HH         XXXXXX       LLLLLL       SET FUNCTION
-------------------------------------------------------------
```



TRUE/COMPLEMENT AND CLEAR/SET (cont'd)
DESCRIPTION

THIS PLE8P8 IS A 6-BIT TRUE/COMPLEMENT AND CLEAR/SET LOGIC FUNCTIONS. THE CONTROL LINES (I1 AND I2) SELECT ONE OF FOUR LOGIC FUNCTIONS FOR THE 6-BIT INPUT DATA (D1-D6) AND THE 6-BIT OUTPUT FUNCTION (Y1-Y6).

WHEN I1 IS FALSE (I1=LOW) THE FUNCTION IS INVERT IF I2 IS FALSE (I2=LOW) OR TRUE IF I2 IS TRUE (I2=HIGH).

WHEN I1 IS TRUE (I1=HIGH) THE FUNCTION IS CLEAR IF I2 IS FALSE (I2=LOW) OR SET IF I2 IS TRUE (I2=HIGH).

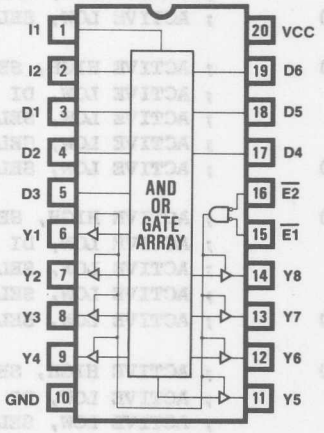
THE PLE8P8 ALSO FEATURES THREE-STATE OUTPUTS WITH TWO ACTIVE LOW OUTPUT ENABLE CONTROLS (/E1 AND /E2).

I1	I2	D1-D6	Y1-Y6	FUNCTION
L	L	D	/D	INVERT
L	H	D	D	TRUE
H	L	X	H	CLEAR
H	H	X	L	SET

4

6-BIT TRUE/COMPLEMENT ZERO/ONE LOGIC FUNCTIONS

PLE8P8



```
PLE5P8                                              PLE DESIGN SPECIFICATION
P5003                                                   FRANK LEE 04/15/84
EXPANDABLE 3-TO-8 DEMULTIPLEXER
MMI SANTA CLARA, CALIFORNIA
.ADD S0 S1 S2 DI PO
.DAT Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7


Y0 =  PO *  DI * /S2 * /S1 * /S0        ; ACTIVE HIGH, SELECT 0
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       *  S2                    ; ACTIVE LOW, SELECT 4-7
   + /PO          *  S1                 ; ACTIVE LOW, SELECT 2,3,6,7
   + /PO             *  S0              ; ACTIVE LOW, SELECT 1,3,5,7

Y1 =  PO *  DI * /S2 * /S1 *  S0        ; ACTIVE HIGH, SELECT 1
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       *  S2                    ; ACTIVE LOW, SELECT 4,5,6,7
   + /PO          *  S1                 ; ACTIVE LOW, SELECT 2,3,6,7
   + /PO             * /S0              ; ACTIVE LOW, SELECT 0,2,4,6

Y2 =  PO *  DI * /S2 *  S1 * /S0        ; ACTIVE HIGH, SELECT 2
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       *  S2                    ; ACTIVE LOW, SELECT 4-7
   + /PO          * /S1                 ; ACTIVE LOW, SELECT 0,1,4,5
   + /PO             *  S0              ; ACTIVE LOW, SELECT 1,3,5,7

Y3 =  PO *  DI * /S2 *  S1 *  S0        ; ACTIVE HIGH, SELECT 3
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       *  S2                    ; ACTIVE LOW, SELECT 4-7
   + /PO          * /S1                 ; ACTIVE LOW, SELECT 0,1,4,5
   + /PO             * /S0              ; ACTIVE LOW, SELECT 0,2,4,6

Y4 =  PO *  DI *  S2 * /S1 * /S0        ; ACTIVE HIGH, SELECT 4
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       * /S2                    ; ACTIVE LOW, SELECT 0-3
   + /PO          *  S1                 ; ACTIVE LOW, SELECT 2,3,6,7
   + /PO             *  S0              ; ACTIVE LOW, SELECT 1,3,5,7

Y5 =  PO *  DI *  S2 * /S1 *  S0        ; ACTIVE HIGH, SELECT 5
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       * /S2                    ; ACTIVE LOW, SELECT 0-3
   + /PO          *  S1                 ; ACTIVE LOW, SELECT 2,3,6,7
   + /PO             * /S0              ; ACTIVE LOW, SELECT 0,2,4,6

Y6 =  PO *  DI *  S2 *  S1 * /S0        ; ACTIVE HIGH, SELECT 6
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       * /S2                    ; ACTIVE LOW, SELECT 0-3
   + /PO          * /S1                 ; ACTIVE LOW, SELECT 0,1,4,5
   + /PO             *  S0              ; ACTIVE LOW, SELECT 1,3,5,7

Y7 =  PO *  DI *  S2 *  S1 *  S0        ; ACTIVE HIGH, SELECT 7
   + /PO *  DI                          ; ACTIVE LOW, DI INACTIVE
   + /PO       * /S2                    ; ACTIVE LOW, SELECT 0-3
   + /PO          * /S1                 ; ACTIVE LOW, SELECT 0,1,4,5
   + /PO             * /S0              ; ACTIVE LOW, SELECT 0,2,4,6
```

EXPANDABLE 3-TO-8 DEMULTIPLEXER (cont'd)

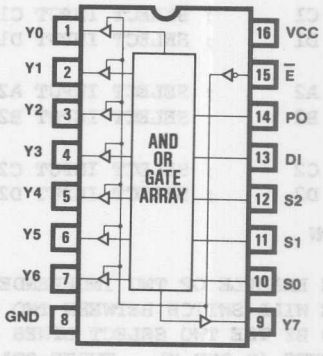
FUNCTION TABLE

PO DI S2 S1 S0 Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0

;		SSS	YYYYYYYY	
;PO	DI	210	76543210	COMMENTS
H	L	XXX	LLLLLLLL	DATA INPUT = 0
H	H	LLL	LLLLLLLH	SELECT OUTPUT 0
H	H	LLH	LLLLLLHL	SELECT OUTPUT 1
H	H	LHL	LLLLLHLL	SELECT OUTPUT 2
H	H	LHH	LLLLHLLL	SELECT OUTPUT 3
H	H	HLL	LLLHLLLL	SELECT OUTPUT 4
H	H	HLH	LLHLLLLL	SELECT OUTPUT 5
H	H	HHL	LHLLLLLL	SELECT OUTPUT 6
H	H	HHH	HLLLLLLL	SELECT OUTPUT 7
L	H	XXX	HHHHHHHH	DATA INPUT = 0
L	L	LLL	HHHHHHHL	SELECT OUTPUT 0
L	L	LLH	HHHHHHLH	SELECT OUTPUT 1
L	L	LHL	HHHHHLHH	SELECT OUTPUT 2
L	L	LHH	HHHHLHHH	SELECT OUTPUT 3
L	L	HLL	HHHLHHHH	SELECT OUTPUT 4
L	L	HLH	HHLHHHHH	SELECT OUTPUT 5
L	L	HHL	HLHHHHHH	SELECT OUTPUT 6
L	L	HHH	LHHHHHHH	SELECT OUTPUT 7

EXPANDABLE 3-TO-8 DEMULTIPLEXER

PLE5P8



DESCRIPTION

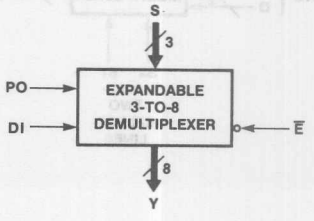
THIS PLE5P8 IMPLEMENTS AN EXPANDABLE 3-TO-8 DEMULTIPLEXER. THE DEVICE DEMULTIPLEXES THREE SELECT INPUT SIGNALS (S2-S0) INTO EIGHT OUTPUTS (Y7-Y0) USING THE INPUT DI WITH POLARITY SELECT PO. SINCE THE DEVICE HAS THREE-STATE OUTPUTS, IT CAN BE EXPANDED USING THE ACTIVE LOW ENABLE PIN (/E).

PIN ASSIGNMENTS:

1. PO — HIGH INDICATES OUTPUT IS ACTIVE HIGH. LOW INDICATES OUTPUT IS ACTIVE LOW.
2. DI — DATA INPUT (DEMULTIPLEXING INPUT). ACTIVE LOW IF PO IS LOW.
3. S2-S0 — SELECT PINS. S2 IS THE MOST SIGNIFICANT BIT. ACTIVE HIGH REGARDLESS OF PO.
4. Y7-Y0 — OUTPUTS. CAN BE ACTIVE HIGH OR ACTIVE LOW DEPENDING ON PO. ACTIVE LOW IF PO IS LOW.

OPERATIONS TABLE:

PO	DI	S2-S0	Y7-Y0	OPERATION
L	H	X	H	OUTPUTS HIGH
H	H	S	DEMUX	DEMUX ACTIVE HIGH
L	L	S	/DEMUX	DEMUX ACTIVE LOW
H	L	X	L	OUTPUTS LOW

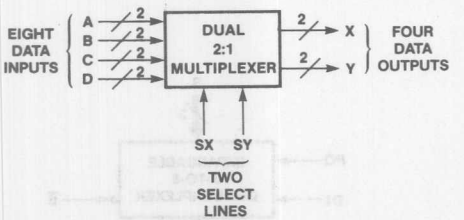


```
DUAL 2:1 MULTIPLEXER
MMI SANTA CLARA, CALIFORNIA
.ADD SX SY A1 B1 C1 D1 A2 B2 C2 D2
.DAT X1 Y1 X2 Y2


X1 = /SX* A1        ; SELECT INPUT A1
   +  SX* B1        ; SELECT INPUT B1

Y1 = /SY* C1        ; SELECT INPUT C1
   +  SY* D1        ; SELECT INPUT D1

X2 = /SX* A2        ; SELECT INPUT A2
   +  SX* B2        ; SELECT INPUT B2

Y2 = /SY* C2        ; SELECT INPUT C2
   +  SY* D2        ; SELECT INPUT D2
```

DESCRIPTION

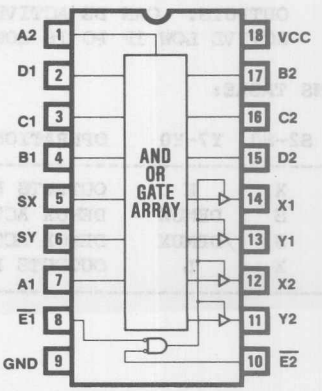
THIS IS AN EXAMPLE OF TWO INDEPENDENT 2-TO-1 MULTIPLEXERS USING A PLE10P4. THE DEVICE WILL SWITCH BETWEEN TWO PAIRS OF 2-BIT INPUTS (A, B AND C, D), AS DETERMINED BY THE TWO SELECT LINES (SX, SY), FOR OUTPUT THROUGH TWO PAIRS OF 2-BIT OUTPUTS (X AND Y). THREE-STATE OUTPUTS ARE ALSO PROVIDED WITH TWO ACTIVE LOW ENABLE PINS (/E1 AND /E2). THE FUNCTIONS OF THE DEVICE ARE SUMMARIZED IN THE TABLE BELOW:

SELECT LINES		INPUT A, B				INPUT C, D					OUTPUT X, Y				
S	S	A	A	B	B	C	C	D	D	!	X	Y	X	Y	
X	Y	1	2	1	2	1	2	1	2	!	1	1	2	2	FUNCTION
L	L	A1	A2	X	X	C1	C2	X	X	!	A1	C1	A2	C2	SELECT A, C
L	H	A1	A2	X	X	X	X	D1	D2	!	A1	D1	A2	D2	SELECT A, D
H	L	X	X	B1	B2	C1	C2	X	X	!	B1	B2	C1	C2	SELECT B, C
H	H	X	X	B1	B2	X	X	D1	D2	!	B1	D1	B2	D2	SELECT B, D



DUAL 2:1 MULTIPLEXER

PLE10P4



```
PLE10P4                                          PLE DESIGN SPECIFICATION
P5005                                                S. HORIKO 04/29/84
QUAD 2:1 MULTIPLEXER WITH POLARITY CONTROL
MMI JAPAN
.ADD SEL POL A0 A1 A2 A3 B0 B1 B2 B3
.DAT Y0 Y1 Y2 Y3

Y0 = /SEL*/POL*/A0                ; SELECT INPUT /A0 (COMP)
   + /SEL* POL* A0                ; SELECT INPUT  A0 (TRUE)
   +  SEL*/POL*/B0                ; SELECT INPUT /B0 (COMP)
   +  SEL* POL* B0                ; SELECT INPUT  B0 (TRUE)

Y1 = /SEL*/POL*/A1                ; SELECT INPUT /A1 (COMP)
   + /SEL* POL* A1                ; SELECT INPUT  A1 (TRUE)
   +  SEL*/POL*/B1                ; SELECT INPUT /B1 (COMP)
   +  SEL* POL* B1                ; SELECT INPUT  B1 (TRUE)

Y2 = /SEL*/POL*/A2                ; SELECT INPUT /A2 (COMP)
   + /SEL* POL* A2                ; SELECT INPUT  A2 (TRUE)
   +  SEL*/POL*/B2                ; SELECT INPUT /B2 (COMP)
   +  SEL* POL* B2                ; SELECT INPUT  B2 (TRUE)

Y3 = /SEL*/POL*/A3                ; SELECT INPUT /A3 (COMP)
   + /SEL* POL* A3                ; SELECT INPUT  A3 (TRUE)
   +  SEL*/POL*/B3                ; SELECT INPUT /B3 (COMP)
   +  SEL* POL* B3                ; SELECT INPUT  B3 (TRUE)


FUNCTION TABLE

SEL POL A0 A1 A2 A3 B0 B1 B2 B3 Y0 Y1 Y2 Y3

; SELECT    AAAA BBBB    YYYY
;SEL  POL   0123 0123    0123      COMMENTS
-------------------------------------------------------------
  L    L    LLLL XXXX    HHHH      SELECT COMP INPUT /A=00
  L    L    LHLH XXXX    HLHL      SELECT COMP INPUT /A=05
  L    L    HLHL XXXX    LHLH      SELECT COMP INPUT /A=10
  L    L    HHHH XXXX    LLLL      SELECT COMP INPUT /A=15
  L    H    LLLL XXXX    LLLL      SELECT TRUE INPUT  A=00
  L    H    LHLH XXXX    LHLH      SELECT TRUE INPUT  A=05
  L    H    HLHL XXXX    HLHL      SELECT TRUE INPUT  A=10
  L    H    HHHH XXXX    HHHH      SELECT TRUE INPUT  A=15
  H    L    XXXX LLLL    HHHH      SELECT COMP INPUT /B=00
  H    L    XXXX LHLH    HLHL      SELECT COMP INPUT /B=05
  H    L    XXXX HLHL    LHLH      SELECT COMP INPUT /B=10
  H    L    XXXX HHHH    LLLL      SELECT COMP INPUT /B=15
  H    H    XXXX LLLL    LLLL      SELECT TRUE INPUT  B=00
  H    H    XXXX HLHL    HLHL      SELECT TRUE INPUT  B=05
  H    H    XXXX LHLH    LHLH      SELECT TRUE INPUT  B=10
  H    H    XXXX HHHH    HHHH      SELECT TRUE INPUT  B=15
-------------------------------------------------------------
```

4

QUAD 2:1 MULTIPLEXER WITH POLARITY CONTROL (cont'd)

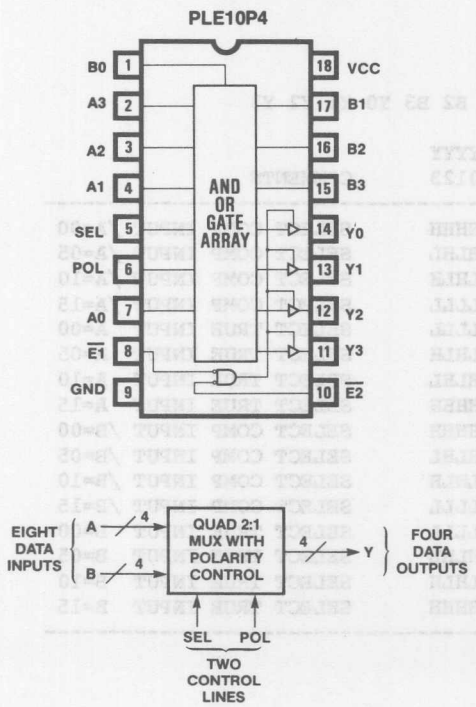
DESCRIPTION

THIS IS AN EXAMPLE OF A QUAD 2:1 MULTIPLEXER WITH POLARITY CONTROL IMPLEMENTED IN A PLE10P4. THE DEVICE SELECTS BETWEEN TWO 4-BIT INPUTS (A1-A4 AND B1-B4) WHICH ARE DIRECTED TO ONE 4-BIT OUTPUT (Y1-Y4) AS DETERMINED BY ONE INPUT SELECT LINE (SEL) AND POLARITY CONTROL (POL). WHEN POLARITY IS TRUE (POL=HIGH), THE TRUE OF THE INPUT SIGNAL IS SELECTED. WHEN POLARITY IS FALSE (POL=LOW), THE COMPLEMENT OF THE INPUT SIGNAL IS SELECTED.

THE PLE10P4 ALSO FEATURES THREE-STATE OUTPUTS WITH TWO ACTIVE LOW ENABLE PINS (/E1 AND /E2). THE FUNCTION IS SUMMARIZED BELOW:

SEL	POL	A1-A4	B1-B4	Y1-Y4
L	H	A	X	A
L	L	A	X	/A
H	H	X	B	B
H	L	X	B	/B

QUAD 2:1 MULTIPLEXER WITH POLARITY CONTROL



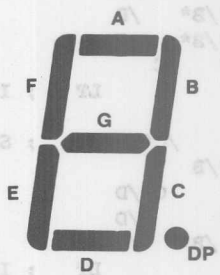
```
PLE5P8                                          PLE DESIGN SPECEFICATION
P5006                                           ULRIK MUELLER 04/29/84
HEXADECIMAL TO SEVEN SEGMENT DECODER
MMI SANTA CLARA, CALIFORNIA
.ADD A B C D LT
.DAT /OA /OB /OC /OD /OE /OF /OG /DP

OA =     B*   /D      ; SEGMENT A
   +     B* C
   + /A*   /C*/D
   +  A*    C*/D
   + /A*       D
   +    /B*/C* D
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT A

OB =       /C*/D      ; SEGMENT B
   +  A* B*   /D
   + /A*/B*   /D
   +  A*/B*    D
   + /A*   /C
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT B

OC =       /C* D      ; SEGMENT C
   +  A*/B
   +       C*/D
   +  A*     /D
   +    /B*  /D
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT C

OD = /A*/B*/C         ; SEGMENT D
   +    /B*    D
   +  A*/B* C
   +  A* B*/C
   + /A* B* C
   + /A* B*  /D
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT D

OE = /A*   /C         ; SEGMENT E
   +       C* D
   + /A* B
   +  A* B*    D
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT E

OF = /A*/B            ; SEGMENT F
   +    /B* C*/D
   +      /C* D
   +     B*    D
   + /A* B* C
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT F

OG =     B*/C         ; SEGMENT G
   + /A* B
   +       /C* D
   +  A*       D
   +    /B* C*/D
   +            LT    ; IF LT=H MAKE BLANK TEST ON SEGMENT G

DP = LT               ; TURNS DP ON ONLY WHEN LT=H
```

DESCRIPTION

THIS EXAMPLE ILLUSTRATES THE USE OF A PLE5P8 AS A HEXADECIMAL TO SEVEN SEGMENT DECODER. THE DEVICE DECODES A 4-BIT BINARY INPUT (D,C,B,A) INTO THE SEVEN SEGMENT OUTPUTS NEEDED TO DRIVE AN LED DISPLAY. NOTE THAT THIS DESIGN IS AN IMPROVEMENT FROM THE 74LS47 SINCE ALL SIXTEEN HEXADECIMAL DIGITS (0-F) CAN BE DISPLAYED. A LAMP TEST IS PROVIDED TO ILLUMINATE ALL SEVEN SEGMENTS AND THE DECIMAL POINT (IF DP IS CONNECTED) BY BRINGING LAMP TEST HIGH (LT=HIGH) REGARDLESS OF THE OTHER BINARY INPUTS. THREE-STATE OUTPUTS ARE ALSO PROVIDED WITH ONE ACTIVE LOW ENABLE PIN (/E).

INPUT DIGIT	LT	INPUT D C B A	SEGMENT ON	OUTPUT DISPLAY
0	L	L L L L	ABCDEF	0
1	L	L L L H	BC	1
2	L	L L H L	ABDEG	2
3	L	L L H H	ABCDG	3
4	L	L H L L	BCDFG	4
5	L	L H L H	ACDFG	5
6	L	L H H L	ACDEFG	6
7	L	L H H H	ABC	7
8	L	H L L L	ABCDEFG	8
9	L	H L L H	ABCFG	9
A	L	H L H L	ABCEFG	A
B	L	H L H H	CDEFG	b
C	L	H H L L	ADEF	C
D	L	H H L H	BCDEG	d
E	L	H H H L	ADEFG	E
F	L	H H H H	AEFG	F
X	H	X X X X	ABCDEFG	8 *

SEGMENT IDENTIFICATION

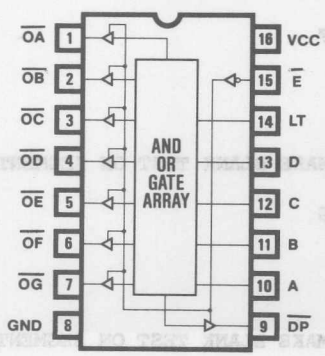


CHARACTER SET

* BLANK TEST OF DISPLAY

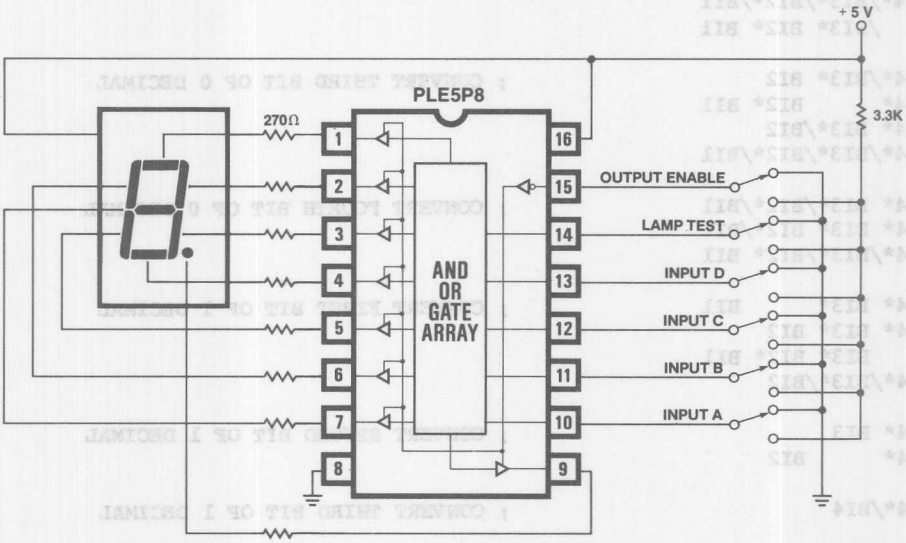
HEXADECIMAL TO SEVEN-SEGMENT DECODER

PLE5P8



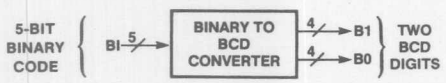
HEXADECIMAL TO SEVEN SEGMENT DECODER (cont'd)

HEXADECIMAL TO SEVEN SEGMENT DECODER

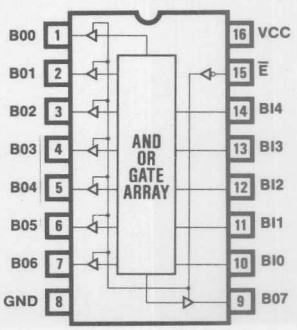


```
PLE5P8                                          PLE DESIGN SPECIFICATION
P5007                                             VINCENT COLI 02/03/82
5-BIT BINARY TO BCD CONVERTER
MMI SANTA CLARA, CALIFORNIA
.ADD BI0 BI1 BI2 BI3 BI4
.DAT B00 B01 B02 B03 B10 B11 B12 B13


B00 =  BI0                                ; CONVERT FIRST BIT OF 0 DECIMAL (LSB)

B01 = /BI4*/BI3*      BI1                 ; CONVERT SECOND BIT OF 0 DECIMAL
    + /BI4* BI3* BI2*/BI1
    +  BI4* BI3*/BI2* BI1
    +  BI4*/BI3*/BI2*/BI1
    +      /BI3* BI2* BI1

B02 = /BI4*/BI3* BI2                      ; CONVERT THIRD BIT OF 0 DECIMAL
    + /BI4*      BI2* BI1
    +  BI4* BI3*/BI2
    +  BI4*/BI3*/BI2*/BI1

B03 = /BI4* BI3*/BI2*/BI1                 ; CONVERT FOURTH BIT OF 0 DECIMAL
    +  BI4* BI3* BI2*/BI1
    +  BI4*/BI3*/BI2* BI1

B10 = /BI4* BI3*      BI1                 ; CONVERT FIRST BIT OF 1 DECIMAL
    + /BI4* BI3* BI2
    +       BI3* BI2* BI1
    +  BI4*/BI3*/BI2

B11 =  BI4* BI3                           ; CONVERT SECOND BIT OF 1 DECIMAL
    +  BI4*       BI2

B12 =  BI4*/BI4                           ; CONVERT THIRD BIT OF 1 DECIMAL

B13 =  BI4*/BI4                           ; CONVERT FOURTH BIT OF 1 DECIMAL (MSB)
```

5-BIT BINARY TO BCD CONVERTER



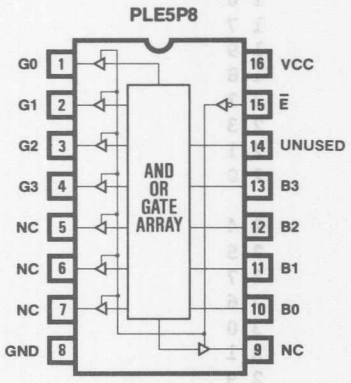
PLE5P8

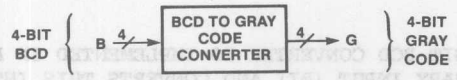


```
PLE5P8                                    PLE DESIGN SPECIFICATION
P5008                                       VINCENT COLI 10/16/81
4-BIT BCD TO GRAY CODE CONVERTER
MMI SANTA CLARA, CALIFORNIA
.ADD B0 B1 B2 B3
.DAT G0 G1 G2 G3


G0 = B0 :+: B1                          ; CONVERT G0 (LSB)

G1 = B1 :+: B2                          ; CONVERT G1

G2 = B2 :+: B3                          ; CONVERT G2

G3 = B3                                 ; CONVERT G3 (MSB)


DESCRIPTION

THIS PLE5P8 WILL CONVERT A 4-BIT BCD INPUT (B3-B0) INTO A 4-BIT
GRAY CODE REPRESENTATION (G3-G0) FOR OUTPUT.
```

4

4-BIT BCD TO GRAY CODE CONVERTER





FUNCTION TABLE

BI4 BI3 BI2 BI1 BI0 B13 B12 B11 B10 B03 B02 B01 B00

;ADDRESS ;BINARY ;43 210	----DATA---- BCD 1 3210	BCD 0 3210	DESCRIPTION (DECIMAL VALUE)
LL LLL	LLLL	LLLL	0
LL LLH	LLLL	LLLH	1
LL LHH	LLLL	LLHH	3
LL LHL	LLLL	LLHL	2
LL HHL	LLLL	LHHL	6
LL HHH	LLLL	LHHH	7
LL HLH	LLLL	LHLH	5
LL HLL	LLLL	LHLL	4
;			
LH LLL	LLLL	HLLL	8
LH LLH	LLLL	HLLH	9
LH LHH	LLLH	LLLH	11
LH LHL	LLLH	LLLL	10
LH HHL	LLLH	LHLL	14
LH HHH	LLLH	LHLH	15
LH HLH	LLLH	LLHH	13
LH HLL	LLLH	LLHL	12
;			
HL LLL	LLLH	LHHL	16
HL LLH	LLLH	LHHH	17
HL LHH	LLLH	HLLH	19
HL LHL	LLLH	HLLL	18
HL HHL	LLHL	LLHL	22
HL HHH	LLHL	LLHH	23
HL HLH	LLHL	LLLH	21
HL HLL	LLHL	LLLL	20
;			
HH LLL	LLHL	LHLL	24
HH LLH	LLHL	LHLH	25
HH LHH	LLHL	LHHH	27
HH LHL	LLHL	LHHL	26
HH HHL	LLHH	LLLL	30
HH HHH	LLHH	LLLH	31
HH HLH	LLHL	HLLH	29
HH HLL	LLHL	HLLL	28

DESCRIPTION

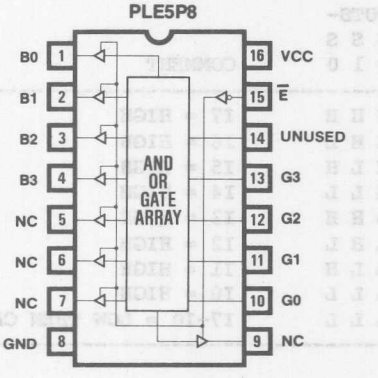
THIS 5-BIT BINARY TO 2-DIGIT BCD CONVERTER IS IMPLEMENTED IN A PLE5P8. THE DEVICE ACCEPTS A 5-BIT BINARY INPUT (BI) AND CONVERTS THIS INTO TWO 4-BIT BINARY CODED DECIMAL (BCD) OUTPUTS (B1 AND B0).

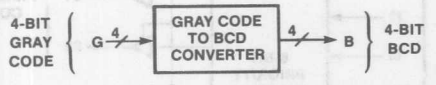
THREE-STATE OUTPUTS ARE ALSO PROVIDED WITH ONE ACTIVE LOW ENABLE PIN (/E).

```
PLE5P8                                  PLE DESIGN SPECIFICATION
P5009                                   VINCENT COLI 03/16/84
4-BIT GRAY CODE TO BCD CONVERTER
MMI SANTA CLARA, CALIFORNIA
.ADD G0 G1 G2 G3
.DAT B0 B1 B2 B3


B0 = G0 :+: G1 :+: G2 :+: G3        ; CONVERT B0 (LSB)

B1 = G1 :+: G2 :+: G3               ; CONVERT B1

B2 = G2 :+: G3                      ; CONVERT B2

B3 = G3                             ; CONVERT B3 (MSB)


DESCRIPTION

THIS PLE5P8 WILL CONVERT A 4-BIT GRAY CODE INPUT (G3-G0) INTO A 4-BIT
BINARY REPRESENTATION (B3-B0) FOR OUTPUT.
```

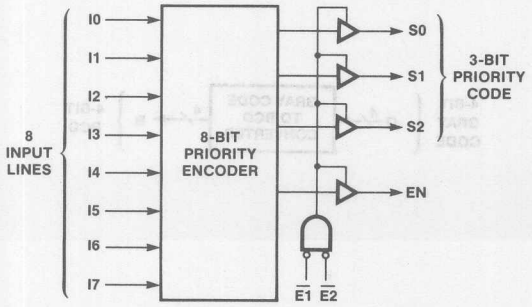
4

4-BIT GRAY CODE TO BCD CONVERTER





```
PLE8P4                                    PLE DESIGN SPECIFICATION
P5010                                FRANK LEE/ULRIK MUELLER 05/14/84
8-BIT PRIORITY ENCODER
MMI SANTA CLARA, CALIFORNIA
.ADD I0 I1 I2 I3 I4 I5 I6 I7
.DAT S0 S1 S2 EN


S0 =  I7                                  ; I7-I0 = 1XXXXXXX
   + /I6* I5                              ; I7-I0 = X01XXXXX
   + /I6*/I4* I3                          ; I7-I0 = X0X01XXX
   + /I6*/I4*/I2* I1                      ; I7-I0 = X0X0X01X

S1 =  I7                                  ; I7-I0 = 1XXXXXXX
   +  I6                                  ; I7-I0 = X1XXXXXX
   + /I5*/I4* I3                          ; I7-I0 = XX001XXX
   + /I5*/I4* I2                          ; I7-I0 = XX00X1XX

S2 =  I7                                  ; I7-I0 = 1XXXXXXX
   +  I6                                  ; I7-I0 = X1XXXXXX
   +  I5                                  ; I7-I0 = XX1XXXXX
   +  I4                                  ; I7-I0 = XXX1XXXX

EN = /I0*/I1*/I2*/I3*/I4*/I5*/I6*/I7     ; ALL LOWS ENABLE NEXT PRIORITY ENCODER


FUNCTION TABLE

I7 I6 I5 I4 I3 I2 I1 I0 EN S2 S1 S0

; --INPUT LINES--      -OUTPUTS-
; I  I  I  I  I  I  I  I    E    S  S  S
; 7  6  5  4  3  2  1  0    N    2  1  0     COMMENT
-------------------------------------------------------------------------
  H  X  X  X  X  X  X  X    L    H  H  H     I7 = HIGH
  L  H  X  X  X  X  X  X    L    H  H  L     I6 = HIGH
  L  L  H  X  X  X  X  X    L    H  L  H     I5 = HIGH
  L  L  L  H  X  X  X  X    L    H  L  L     I4 = HIGH
  L  L  L  L  H  X  X  X    L    L  H  H     I3 = HIGH
  L  L  L  L  L  H  X  X    L    L  H  L     I2 = HIGH
  L  L  L  L  L  L  H  X    L    L  L  H     I1 = HIGH
  L  L  L  L  L  L  L  H    L    L  L  L     I0 = HIGH
  L  L  L  L  L  L  L  L    H    L  L  L     I7-I0 = LOW THEN CARRY OUT
-------------------------------------------------------------------------
```



8-BIT PRIORITY ENCODER (cont'd)

DESCRIPTION

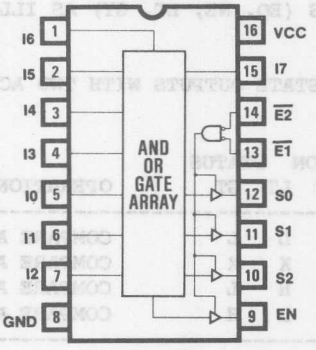
THIS 8-BIT PRIORITY ENCODER SCANS FOR THE FIRST HIGH INPUT LINE (I7-I0) FROM I7 (WHICH HAS THE HIGHEST PRIORITY) TO I0 (WHICH HAS THE LOWEST PRIORITY). IT WILL GENERATE A BINARY ENCODED OUTPUT (S2-S0) WHICH WILL POINT TO THE HIGHEST PRIORITY INPUT WHICH IS AT A HIGH STATE.

IF NO INPUT LINES ARE HIGH (I7-I0=LOW), THEN THE BINARY ENCODED OUTPUTS WILL BE ZERO (S2-S0=LOW) AND THE ENABLE OUTPUT WILL BE HIGH (EN=HIGH) INDICATING A CARRY OUT TO THE NEXT PRIORITY ENCODER. THE OUTPUT ENABLE WILL BE LOW (EN=LOW) IF ANY OF THE INPUT LINES ARE HIGH.

THE PLE8P4 ALSO HAS THREE-STATE OUTPUTS WITH TWO ACTIVE-LOW OUTPUT ENABLE CONTROL PINS (/E1 AND /E2).

8-BIT PRIORITY ENCODER

PLE8P4



```
4-BIT MAGNITUDE COMPARATOR
MMI SANTA CLARA, CALIFORNIA
.ADD A0 A1 A2 A3 B0 B1 B2 B3
.DAT EQ NE LT GT

EQ =  A3:*:B3 *  A2:*:B2 *  A1:*:B1 *  A0:*:B0 ; A = B

NE =  A3:+:B3 +  A2:+:B2 +  A1:+:B1 +  A0:+:B0 ; A NOT = B

LT = /A3 * B3                                   ; A3 LT B3
   +  A3:*:B3 * /A2 * B2                        ; A2 LT B2
   +  A3:*:B3 *  A2:*:B2 * /A1 * B1             ; A1 LT B1
   +  A3:*:B3 *  A2:*:B2 *  A1:*:B1 * /A0 * B0 ; A0 LT B0

GT =  A3 */B3                                   ; A3 GT B3
   +  A3:*:B3 *  A2 */B2                        ; A2 GT B2
   +  A3:*:B3 *  A2:*:B2 *  A1 */B1             ; A1 GT B1
   +  A3:*:B3 *  A2:*:B2 *  A1:*:B1 *  A0 */B0 ; A0 GT B0
```

DESCRIPTION

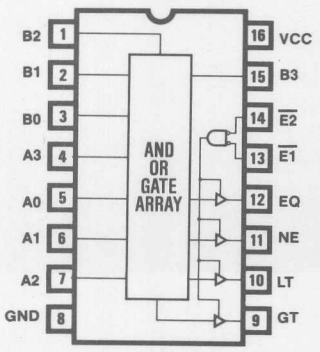
THIS PLE8P4 COMPARES TWO 4-BIT NUMBERS (A3-A0 AND B3-B0) TO ESTABLISH IF THEY ARE EQUAL (A = B THEN EQ=H), NOT EQUAL (A NOT = B THEN NE=H), LESS THAN (A LT B THEN LT=H), OR GREATER THAN (A GT B THEN GT=H) AND REPORTS THE COMPARISON STATUS ON THE OUTPUTS (EQ, NE, LT, GT) AS ILLUSTRATED IN THE OPERATIONS TABLE BELOW.

THE PLE8P4 ALSO FEATURES THREE-STATE OUTPUTS WITH TWO ACTIVE-LOW OUTPUT ENABLE CONTROL PINS (/E1 AND /E2).

INPUT NUMBERS A3-A0	B3-B0	COMPARISON STATUS EQ	NE	LT	GT	OPERATION
A	= B	H	L	L	L	COMPARE A EQUAL TO B
A NOT =	B	L	H	X	X	COMPARE A NOT EQUAL TO B
A LT	B	L	H	H	L	COMPARE A LESS THAN B
A GT	B	L	H	L	H	COMPARE A GREATER THAN B

4-BIT MAGNITUDE COMPARATOR

PLE8P4



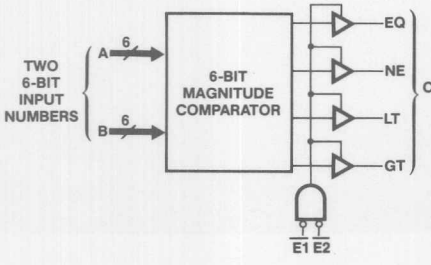
```
PLE12P4                                          PLE DESIGN SPECIFICATION
P5012                                              VINCENT COLI 10/16/83
6-BIT MAGNITUDE COMPARATOR
MMI SANTA CLARA, CALIFORNIA
.ADD A0 A1 A2 A3 A4 A5 B0 B1 B2 B3 B4 B5
.DAT EQ NE LT GT

EQ =  A5:*:B5 *  A4:*:B4 *  A3:*:B3 *  A2:*:B2 *  A1:*:B1 *  A0:*:B0 ; A = B

NE =  A5:+:B5 +  A4:+:B4 +  A3:+:B3 +  A2:+:B2 +  A1:+:B1 +  A0:+:B0 ; A NOT= B

LT = /A5 * B5                                                          ; A5 LT B5
   +  A5:*:B5 * /A4 * B4                                               ; A4 LT B4
   +  A5:*:B5 *  A4:*:B4 * /A3 * B3                                    ; A3 LT B3
   +  A5:*:B5 *  A4:*:B4 *  A3:*:B3 * /A2 * B2                         ; A2 LT B2
   +  A5:*:B5 *  A4:*:B4 *  A3:*:B3 *  A2:*:B2 * /A1 * B1              ; A1 LT B1
   +  A5:*:B5 *  A4:*:B4 *  A3:*:B3 *  A2:*:B2 *  A1:*:B1 * /A0 * B0 ; A0 LT B0

GT =  A5 */B5                                                          ; A5 GT B5
   +  A5:*:B5 *  A4 */B4                                               ; A4 GT B4
   +  A5:*:B5 *  A4:*:B4 *  A3 */B3                                    ; A3 GT B3
   +  A5:*:B5 *  A4:*:B4 *  A3:*:B3 *  A2 */B2                         ; A2 GT B2
   +  A5:*:B5 *  A4:*:B4 *  A3:*:B3 *  A2:*:B2 *  A1 */B1              ; A1 GT B1
   +  A5:*:B5 *  A4:*:B4 *  A3:*:B3 *  A2:*:B2 *  A1:*:B1 *  A0 */B0 ; A0 GT B0
```

DESCRIPTION

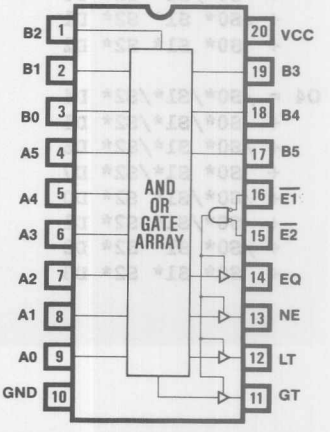
THIS PLE12P4 COMPARES TWO 6-BIT NUMBERS (A5-A0 AND B5-B0) TO ESTABLISH IF THEY ARE EQUAL (A = B THEN EQ=H), NOT EQUAL (A NOT = B THEN NE=H), LESS THAN (A LT B THEN LT=H), OR GREATER THAN (A GT B THEN GT=H) AND REPORTS THE COMPARISON STATUS ON THE OUTPUTS (EQ, NE, LT, GT) AS ILLUSTRATED IN THE OPERATIONS TABLE BELOW.

THE PLE12P4 ALSO FEATURES THREE-STATE OUTPUTS WITH TWO ACTIVE-LOW OUTPUT ENABLE CONTROL PINS (/E1 AND /E2).

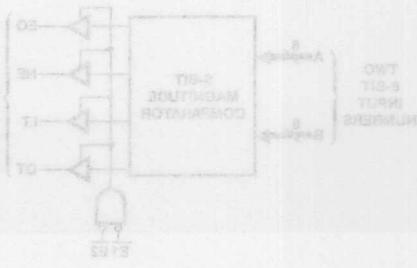
INPUT NUMBERS A5-A0	B5-B0	COMPARISON STATUS EQ	NE	LT	GT	OPERATION
A	= B	H	L	L	L	COMPARE A EQUAL TO B
A NOT	= B	L	H	X	X	COMPARE A NOT EQUAL TO B
A LT	B	L	H	H	L	COMPARE A LESS THAN B
A GT	B	L	H	L	H	COMPARE A GREATER THAN B



6-BIT MAGNITUDE COMPARATOR
PLE12P4



```
PLE11P8                                    PLE DESIGN SPECIFICATION
P5013                                        VINCENT COLI 06/12/84
8-BIT BARREL SHIFTER
MMI SANTA CLARA, CALIFORNIA
.ADD D0 D1 D2 D3 D4 D5 D6 D7 S0 S1 S2
.DAT O0 O1 O2 O3 O4 O5 O6 O7

O0 = /S0*/S1*/S2* D0      ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D1      ; SHIFT 1 PLACES
   + /S0* S1*/S2* D2      ; SHIFT 2 PLACES
   +  S0* S1*/S2* D3      ; SHIFT 3 PLACES
   + /S0*/S1* S2* D4      ; SHIFT 4 PLACES
   +  S0*/S1* S2* D5      ; SHIFT 5 PLACES
   + /S0* S1* S2* D6      ; SHIFT 6 PLACES
   +  S0* S1* S2* D7      ; SHIFT 7 PLACES

O1 = /S0*/S1*/S2* D1      ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D2      ; SHIFT 1 PLACES
   + /S0* S1*/S2* D3      ; SHIFT 2 PLACES
   +  S0* S1*/S2* D4      ; SHIFT 3 PLACES
   + /S0*/S1* S2* D5      ; SHIFT 4 PLACES
   +  S0*/S1* S2* D6      ; SHIFT 5 PLACES
   + /S0* S1* S2* D7      ; SHIFT 6 PLACES
   +  S0* S1* S2* D0      ; SHIFT 7 PLACES

O2 = /S0*/S1*/S2* D2      ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D3      ; SHIFT 1 PLACES
   + /S0* S1*/S2* D4      ; SHIFT 2 PLACES
   +  S0* S1*/S2* D5      ; SHIFT 3 PLACES
   + /S0*/S1* S2* D6      ; SHIFT 4 PLACES
   +  S0*/S1* S2* D7      ; SHIFT 5 PLACES
   + /S0* S1* S2* D0      ; SHIFT 6 PLACES
   +  S0* S1* S2* D1      ; SHIFT 7 PLACES

O3 = /S0*/S1*/S2* D3      ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D4      ; SHIFT 1 PLACES
   + /S0* S1*/S2* D5      ; SHIFT 2 PLACES
   +  S0* S1*/S2* D6      ; SHIFT 3 PLACES
   + /S0*/S1* S2* D7      ; SHIFT 4 PLACES
   +  S0*/S1* S2* D0      ; SHIFT 5 PLACES
   + /S0* S1* S2* D1      ; SHIFT 6 PLACES
   +  S0* S1* S2* D2      ; SHIFT 7 PLACES

O4 = /S0*/S1*/S2* D4      ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D5      ; SHIFT 1 PLACES
   + /S0* S1*/S2* D6      ; SHIFT 2 PLACES
   +  S0* S1*/S2* D7      ; SHIFT 3 PLACES
   + /S0*/S1* S2* D0      ; SHIFT 4 PLACES
   +  S0*/S1* S2* D1      ; SHIFT 5 PLACES
   + /S0* S1* S2* D2      ; SHIFT 6 PLACES
   +  S0* S1* S2* D3      ; SHIFT 7 PLACES
```



8-BIT BARREL SHIFTER (cont'd)

```
O5 = /S0*/S1*/S2* D5       ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D6       ; SHIFT 1 PLACES
   + /S0* S1*/S2* D7       ; SHIFT 2 PLACES
   +  S0* S1*/S2* D0       ; SHIFT 3 PLACES
   + /S0*/S1* S2* D1       ; SHIFT 4 PLACES
   +  S0*/S1* S2* D2       ; SHIFT 5 PLACES
   + /S0* S1* S2* D3       ; SHIFT 6 PLACES
   +  S0* S1* S2* D4       ; SHIFT 7 PLACES

O6 = /S0*/S1*/S2* D6       ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D7       ; SHIFT 1 PLACES
   + /S0* S1*/S2* D0       ; SHIFT 2 PLACES
   +  S0* S1*/S2* D1       ; SHIFT 3 PLACES
   + /S0*/S1* S2* D2       ; SHIFT 4 PLACES
   +  S0*/S1* S2* D3       ; SHIFT 5 PLACES
   + /S0* S1* S2* D4       ; SHIFT 6 PLACES
   +  S0* S1* S2* D5       ; SHIFT 7 PLACES

O7 = /S0*/S1*/S2* D7       ; SHIFT 0 PLACES
   +  S0*/S1*/S2* D0       ; SHIFT 1 PLACES
   + /S0* S1*/S2* D1       ; SHIFT 2 PLACES
   +  S0* S1*/S2* D2       ; SHIFT 3 PLACES
   + /S0*/S1* S2* D3       ; SHIFT 4 PLACES
   +  S0*/S1* S2* D4       ; SHIFT 5 PLACES
   + /S0* S1* S2* D5       ; SHIFT 6 PLACES
   +  S0* S1* S2* D6       ; SHIFT 7 PLACES
```

FUNCTION TABLE

S2 S1 S0 D7 D6 D5 D4 D3 D2 D1 D0 O7 O6 O5 O4 O3 O2 O1 O0

;SHIFT ; SSS ; 210	INPUT DATA DDDDDDDD 76543210	OUTPUT DATA OOOOOOOO 76543210	COMMENTS
LLL	HLLLLLLL	HLLLLLLL	BARREL SHIFT ONE HIGH 0 PLACES
LLH	HLLLLLLL	LHLLLLLL	BARREL SHIFT ONE HIGH 1 PLACES
LHL	HLLLLLLL	LLHLLLLL	BARREL SHIFT ONE HIGH 2 PLACES
LHH	HLLLLLLL	LLLHLLLL	BARREL SHIFT ONE HIGH 3 PLACES
HLL	HLLLLLLL	LLLLHLLL	BARREL SHIFT ONE HIGH 4 PLACES
HLH	HLLLLLLL	LLLLLHLL	BARREL SHIFT ONE HIGH 5 PLACES
HHL	HLLLLLLL	LLLLLLHL	BARREL SHIFT ONE HIGH 6 PLACES
HHH	HLLLLLLL	LLLLLLLH	BARREL SHIFT ONE HIGH 7 PLACES
LLL	LHHHHHHH	LHHHHHHH	BARREL SHIFT ONE LOW 0 PLACES
LLH	LHHHHHHH	HLHHHHHH	BARREL SHIFT ONE LOW 1 PLACES
LHL	LHHHHHHH	HHLHHHHH	BARREL SHIFT ONE LOW 2 PLACES
LHH	LHHHHHHH	HHHLHHHH	BARREL SHIFT ONE LOW 3 PLACES
HLL	LHHHHHHH	HHHHLHHH	BARREL SHIFT ONE LOW 4 PLACES
HLH	LHHHHHHH	HHHHHLHH	BARREL SHIFT ONE LOW 5 PLACES
HHL	LHHHHHHH	HHHHHHLH	BARREL SHIFT ONE LOW 6 PLACES
HHH	LHHHHHHH	HHHHHHHL	BARREL SHIFT ONE LOW 7 PLACES

8-BIT BARREL SHIFTER (cont'd)

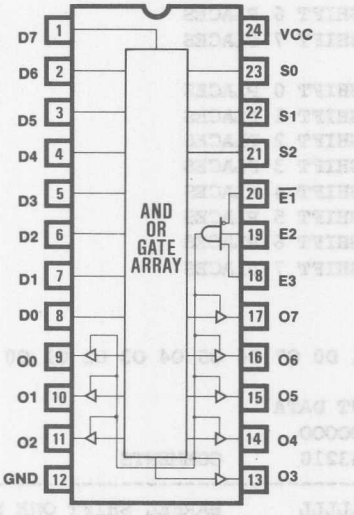
DESCRIPTION

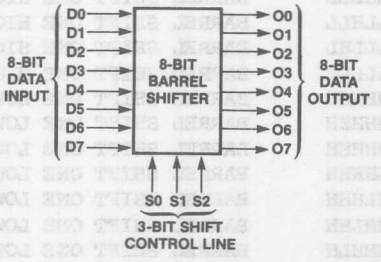
THE 8-BIT BARREL SHIFTER, IMPLEMENTED IN A PLE11P8, ROTATES EIGHT BITS OF DATA (D7-D0) A NUMBER OF LOCATIONS INTO THE OUTPUTS (O7-O0) AS SPECIFIED BY THE 3-BIT BINARY ENCODED SHIFT CONTROL LINE (S2-S0). THE THREE-STATE OUTPUTS ARE IN A HIGH-Z STATE WHEN ANY ONE OF THE TWO OUTPUT ENABLE PINS (/E1 OR /E1) ARE HIGH.

A POSSIBLE UPGRADE VERSION OF THIS DESIGN IMPLEMENTED IN A PLE12P8 COULD INCLUDE A DIRECTION CONTROL LINE. THIS CONTROL LINE PERMITS THE 8-BIT BARREL SHIFTER TO ROTATE DATA IN EITHER DIRECTION (LEFT OR RIGHT).

8-BIT BARREL SHIFTER

PLE11P8





```
PLE11P4                                          PLE DESIGN SPECIFICATION
P5014                                                 CHRIS JAY 05/30/84
4-BIT RIGHT SHIFTER WITH PROGRAMMABLE OUTPUT POLARITY
MMI LTD., FARNBOROUGH, U.K.
.ADD S0 S1 INV D0 D1 D2 D3 D4 D5 D6 /EN
.DAT O0 O1 O2 O3


O0 =  D0*/S0*/S1*/INV* EN      ; SELECT INPUT  D0
   + /D0*/S0*/S1* INV* EN      ; SELECT INPUT /D0
   +  D1* S0*/S1*/INV* EN      ; SELECT INPUT  D1
   + /D1* S0*/S1* INV* EN      ; SELECT INPUT /D1
   +  D2*/S0* S1*/INV* EN      ; SELECT INPUT  D2
   + /D2*/S0* S1* INV* EN      ; SELECT INPUT /D2
   +  D3* S0* S1*/INV* EN      ; SELECT INPUT  D3
   + /D3* S0* S1* INV* EN      ; SELECT INPUT /D3

O1 =  D1*/S0*/S1*/INV* EN      ; SELECT INPUT  D1
   + /D1*/S0*/S1* INV* EN      ; SELECT INPUT /D1
   +  D2* S0*/S1*/INV* EN      ; SELECT INPUT  D2
   + /D2* S0*/S1* INV* EN      ; SELECT INPUT /D2
   +  D3*/S0* S1*/INV* EN      ; SELECT INPUT  D3
   + /D3*/S0* S1* INV* EN      ; SELECT INPUT /D3
   +  D4* S0* S1*/INV* EN      ; SELECT INPUT  D4
   + /D4* S0* S1* INV* EN      ; SELECT INPUT /D4

O2 =  D2*/S0*/S1*/INV* EN      ; SELECT INPUT  D2
   + /D2*/S0*/S1* INV* EN      ; SELECT INPUT /D2
   +  D3* S0*/S1*/INV* EN      ; SELECT INPUT  D3
   + /D3* S0*/S1* INV* EN      ; SELECT INPUT /D3
   +  D4*/S0* S1*/INV* EN      ; SELECT INPUT  D4
   + /D4*/S0* S1* INV* EN      ; SELECT INPUT /D4
   +  D5* S0* S1*/INV* EN      ; SELECT INPUT  D5
   + /D5* S0* S1* INV* EN      ; SELECT INPUT /D5

O3 =  D3*/S0*/S1*/INV* EN      ; SELECT INPUT  D3
   + /D3*/S0*/S1* INV* EN      ; SELECT INPUT /D3
   +  D4* S0*/S1*/INV* EN      ; SELECT INPUT  D4
   + /D4* S0*/S1* INV* EN      ; SELECT INPUT /D4
   +  D5*/S0* S1*/INV* EN      ; SELECT INPUT  D5
   + /D5*/S0* S1* INV* EN      ; SELECT INPUT /D5
   +  D6* S0* S1*/INV* EN      ; SELECT INPUT  D6
   + /D6* S0* S1* INV* EN      ; SELECT INPUT /D6
```

```
/EN S1 S0 INV D6 D5 D4 D3 D2 D1 D0 O3 O2 O1 O0

;-CONTROL-
;/       I    -INPUT  DATA-    OUTPUTS
;E  S S  N    D D D D D D D    O O O O
;N  1 0  V    6 5 4 3 2 1 0    3 2 1 0     COMMENTS
-------------------------------------------------------------------------
 H  X X  X    X X X X X X X    L L L L    TEST ENABLE, OUTPUTS GO LOW
 L  L L  L    L L L H H H H    H H H H    SHIFT COUNT = 0, TRUE POLARITY
 L  L H  L    L L L H H H H    L H H H    SHIFT COUNT = 1, TRUE POLARITY
 L  H L  L    L L L H H H H    L L H H    SHIFT COUNT = 2, TRUE POLARITY
 L  H H  L    L L L H H H H    L L L H    SHIFT COUNT = 3, TRUE POLARITY
 L  L L  H    L L L H H H H    L L L L    SHIFT COUNT = 0, COMP POLARITY
 L  L H  H    L L L H H H H    H L L L    SHIFT COUNT = 1, COMP POLARITY
 L  H L  H    L L L H H H H    H H L L    SHIFT COUNT = 2, COMP POLARITY
 L  H H  H    L L L H H H H    H H H L    SHIFT COUNT = 3, COMP POLARITY
-------------------------------------------------------------------------
```

4-BIT RIGHT SHIFTER WITH PROGRAMMABLE OUTPUT POLARITY (cont'd)

DESCRIPTION

THIS PLE11P4 IMPLEMENTS A 4-BIT RIGHT SHIFTER WITH PROGRAMMABLE OUTPUT POLARITY. THE SHIFTER CAN RIGHT SHIFT SEVEN BITS OF DATA, FOUR BITS AT A TIME. THE SEVEN DATA INPUTS (D6-D0) ARE SHIFTED 0, 1, 2, OR 3 LOCATIONS AS DETERMINED BY THE 2-BIT SHIFT CONTROL LINE (S1-S0). THE SHIFTED DATA IS THEN DIRECTED TO THE FOUR OUTPUTS (O3-O0).

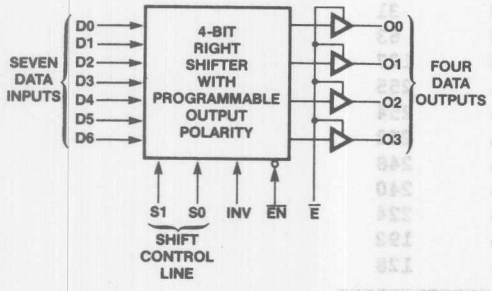
THE OUTPUT DATA IS NONINVERTED (O=D) WHEN INV=L AND INVERTED (O=/D) WHEN INV=H. THE OUTPUTS ARE FORCED LOW (O=L) WHEN /EN=H REGARDLESS OF OTHER INPUTS. THE PLE11P4 ALSO FEATURES THREE-STATE OUTPUTS WITH ONE ACTIVE LOW OUTPUT ENABLE (/E).

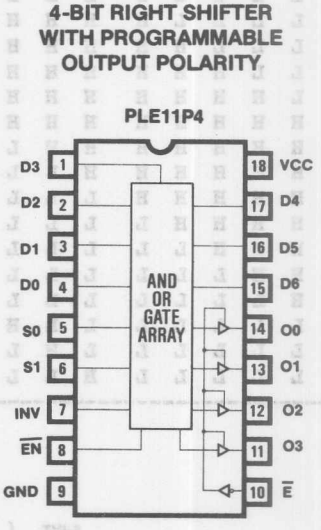
A POSSIBLE UPGRADE VERSION OF THIS DESIGN IMPLEMENTED IN A PLE12P4 COULD INCLUDE A DIRECTION CONTROL LINE. THIS CONTROL LINE PERMITS THE 4-BIT RIGHT SHIFTER TO SHIFT DATA IN EITHER DIRECTION (LEFT OR RIGHT).

OPERATIONS TABLE:

/EN	INV	S1-S0	D6-D0	O3-O0	OPERATION
H	X	X	X	L	DISABLE OUTPUTS LOW
L	L	N	D	SHIFT(D)	SHIFT NONINVERTED DATA "N" PLACES
L	H	N	D	SHIFT(/D)	SHIFT INVERTED DATA "N" PLACES

4

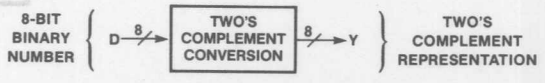




```
PLE8P8                                          PLE DESIGN SPECIFICATION
P5015                                             MIKE VOGEL 11/28/83
8-BIT TWO'S COMPLEMENT CONVERSION
MMI BREA, CALIFORNIA
.ADD D0 D1 D2 D3 D4 D5 D6 D7
.DAT Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7

Y0 = D0                                          ; CONVERT 1ST BIT (LSB)

Y1 = D1 :+: D0                                   ; CONVERT 2ND BIT

Y2 = D2 :+: D0 + D1                              ; CONVERT 3RD BIT

Y3 = D3 :+: D0 + D1 + D2                         ; CONVERT 4TH BIT

Y4 = D4 :+: D0 + D1 + D2 + D3                    ; CONVERT 5TH BIT

Y5 = D5 :+: D0 + D1 + D2 + D3 + D4               ; CONVERT 6TH BIT

Y6 = D6 :+: D0 + D1 + D2 + D3 + D4 + D5          ; CONVERT 7TH BIT

Y7 = D7 :+: D0 + D1 + D2 + D3 + D4 + D5 + D6     ; CONVERT 8TH BIT (MSB)
```

FUNCTION TABLE

D7	D6	D5	D4	D3	D2	D1	D0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	;DECIMAL
L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L	0
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	1
L	L	L	L	L	L	H	H	H	H	H	H	H	H	L	H	3
L	L	L	L	L	H	H	H	H	H	H	H	H	L	L	H	7
L	L	L	L	H	H	H	H	H	H	H	H	L	L	L	H	15
L	L	L	H	H	H	H	H	H	H	H	L	L	L	L	H	31
L	L	H	H	H	H	H	H	H	H	L	L	L	L	L	H	63
L	H	H	H	H	H	H	H	H	L	L	L	L	L	L	H	127
H	H	H	H	H	H	H	H	L	L	L	L	L	L	L	L	255
H	H	H	H	H	H	H	L	L	L	L	L	L	L	H	L	254
H	H	H	H	H	H	L	L	L	L	L	L	L	H	L	L	252
H	H	H	H	H	L	L	L	L	L	L	L	H	L	L	L	248
H	H	H	H	L	L	L	L	L	L	L	H	L	L	L	L	240
H	H	H	L	L	L	L	L	L	L	H	L	L	L	L	L	224
H	H	L	L	L	L	L	L	L	H	L	L	L	L	L	L	192
H	L	L	L	L	L	L	L	H	L	L	L	L	L	L	L	128



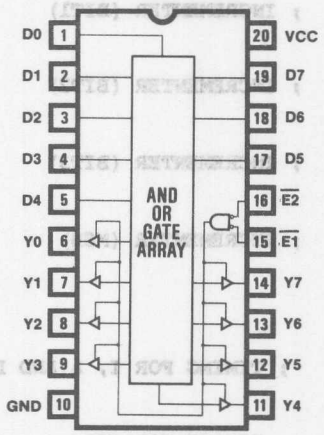
8-BIT TWO'S COMPLEMENT CONVERSION (cont'd)

DESCRIPTION

THIS PLE8P8 CONVERTS AN 8-BIT BINARY NUMBER (D7-D0) INTO TWO'S COMPLEMENT REPRESENTATION (Y7-Y0) WHERE D7 AND Y7 ARE THE MSB AND D0 AND Y0 ARE THE LSB. TWO'S COMPLEMENT REPRESENTATION IS USED IN SIGNED ARITHMETIC SYSTEMS.

8-BIT TWO'S COMPLEMENT CONVERSION

PLE8P8



4

```
A PORTION OF TIMING GENERATOR FOR PAL ARRAY PROGRAMMING
MMI JAPAN
.ADD A0 A1 A2 A3 A4
.DAT NA0 NA1 NA2 NA3 NA4 TIALR TVCC TO

; NEXT ADDRESS GENERATOR

NA0  =  /A0                          ; INCREMENTER (LSB)

NA1  =   A0                          ; INCREMENTER (BIT1)
    :+:  A1

NA2  =   A2                          ; INCREMENTER (BIT2)
    :+:  A0* A1

NA3  =   A3                          ; INCREMENTER (BIT3)
    :+:  A0* A1* A2

NA4  =   A4                          ; INCREMENTER (MSB)
    :+:  A0* A1* A2* A3

; TIMING WAVEFORMS

TIALR = /A4*/A3                      ; TIMING FOR I, A AND L/R
      + /A4*    /A2*/A1

TVCC  = /A4*/A3* A2                  ; TIMING FOR VCC
      + /A4* A3*/A2*/A1

TO    = /A4* A3*/A2*/A1*/A0          ; TIMING FOR O
      + /A4*/A3* A2* A1
      + /A4*/A3* A2*     A0
```

TIMING GENERATOR FOR PAL PROGRAMMING (cont'd)

FUNCTION TABLE

```
A4 A3 A2 A1 A0 NA4 NA3 NA2 NA1 NA0 TIALR TVCC TO

;          NNNNN
;AAAAA      AAAAA    TIMING WAVEFORMS
;43210      43210    TIALR  TVCC   TO      ; ## ;     COMMENTS
-------------------------------------------------------------------------
 LLLLL      LLLLH      H      L      L      ; 01 ;     ASSERT TIALR
 LLLLH      LLLHL      H      L      L      ; 02 ;
 LLLHL      LLLHH      H      L      L      ; 03 ;
 LLLHH      LLHLL      H      L      L      ; 04 ;
 LLHLL      LLHLH      H      H      L      ; 05 ;     ASSERT TVCC
 LLHLH      LLHHL      H      H      H      ; 06 ;     ASSERT TO
 LLHHL      LLHHH      H      H      H      ; 07 ;
 LLHHH      LHLLL      H      H      H      ; 08 ;
 LHLLL      LHLLH      H      H      H      ; 09 ;
 LHLLH      LHLHL      H      H      L      ; 10 ;     CLEAR TO
 LHLHL      LHLHH      L      L      L      ; 11 ;     CLEAR TIALR & TVCC
 LHLHH      LHHLL      L      L      L      ; 12 ;
 LHHLL      LHHLH      L      L      L      ; 13 ;
 LHHLH      LHHHL      L      L      L      ; 14 ;
 LHHHL      LHHHH      L      L      L      ; 15 ;
 LHHHH      HLLLL      L      L      L      ; 16 ;
 HLLLL      HLLLH      L      L      L      ; 17 ;
 HLLLH      HLLHL      L      L      L      ; 18 ;
 HLLHL      HLLHH      L      L      L      ; 19 ;
 HLLHH      HLHLL      L      L      L      ; 20 ;
 HLHLL      HLHLH      L      L      L      ; 21 ;
 HLHLH      HLHHL      L      L      L      ; 22 ;
 HLHHL      HLHHH      L      L      L      ; 23 ;
 HLHHH      HHLLL      L      L      L      ; 24 ;
 HHLLL      HHLLH      L      L      L      ; 25 ;
 HHLLH      HHLHL      L      L      L      ; 26 ;
 HHLHL      HHLHH      L      L      L      ; 27 ;
 HHLHH      HHHLL      L      L      L      ; 28 ;
 HHHLL      HHHLH      L      L      L      ; 29 ;
 HHHLH      HHHHL      L      L      L      ; 30 ;
 HHHHL      HHHHH      L      L      L      ; 31 ;
 HHHHH      LLLLL      L      L      L      ; 32 ;
-------------------------------------------------------------------------
```

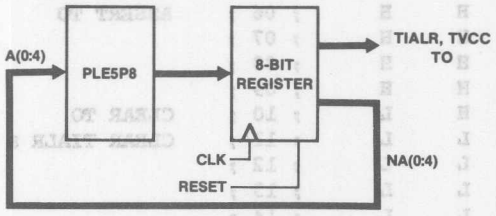
TIMING GENERATOR FOR PAL SECURITY FUSE PROGRAMMING (cont'd)

DESCRIPTION

THIS LOGIC SPECIFICATION IS A TIMING SIGNAL GENERATOR TO BE USED FOR ARRAY PROGRAMMING OF PAL DEVICES. A PLE8P8 FOLLOWED BY A REGISTERED ARE USED TO IMPLEMENT THIS FUNCTION.

THE PLE CONTAINS BOTH 5-BIT NEXT ADDRESS AND 3-BIT WAVEFORMS. TIALR OUTPUT IS A TIMING WAVEFORM FOR I, A, AND L/R SIGNALS, AND TVCC AND TO OUTPUTS ARE USED FOR VCC AND O SIGNALS, RESPECTIVELY. (SEE "PAL PROGRAMMING/VERIFYING PROCEDURE" IN THE PAL DATA SHEET FOR MORE DETAILS)

THE SCHEMATIC IS AS FOLLOWS:

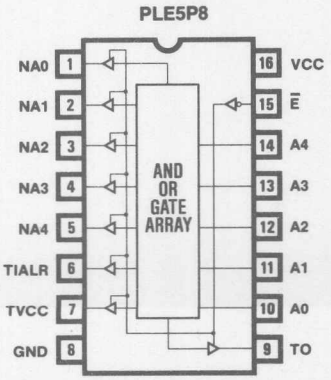


APPLYING 200KHz CLOCK SIGNAL TO THE CLK INPUT OF THE REGISTER GENERATES THE FOLLOWING TIMINGS:

1. I, A, AND L/R WIDTH : 50 usec
2. tD2 : 20 usec
3. tD : 5 usec
4. tVCCP : 30 usec
5. Tp : 20 usec

BECAUSE THE TIMING PATTERNS ARE STORED IN THE PLE, WE CAN EASILY CALIBRATE THE RELATIONS AND THE PERIOD AMONG THOSE SIGNALS TO MAKE AN OPTIMUN CONDITION.

A PORTION OF A
TIMING GENERATOR FOR
PAL ARRAY PROGRAMMING



```
PLE5P8                                    PLE DESIGN SPECIFICATION
P5028                                          S. HORIKO 11/29/83
TIMING GENERATOR FOR PAL SECURITY FUSE PROGRAMMING
MMI JAPAN
.ADD A0 A1 A2 A3 A4
.DAT NA0 NA1 NA2 NA3 NA4 TVCC TP01 TP11

; NEXT ADDRESS GENERATOR
;  ( THE COUNTER LOCKS UP AT COUNT-22 )

NA0  = /A4*          /A1*/A0          ; INCREMENTER (LSB)
     + /A4*           A1*/A0          ; INCREMENTER (LSB)
     +  A4*/A3*/A2*     /A0           ; INCREMENTER (LSB)
     +  A4*/A3* A2*/A1                ; INCREMENTER (LSB)

NA1  = /A4*          /A1* A0          ; INCREMENTER (BIT1)
     + /A4*           A1*/A0          ; INCREMENTER (BIT1)
     +  A4*/A3*/A2*/A1* A0            ; INCREMENTER (BIT1)
     +  A4*/A3*/A2* A1*/A0             ; INCREMENTER (BIT1)

NA2  = /A4*      A2*/A1               ; INCREMENTER (BIT2)
     + /A4*      A2*     /A0          ; INCREMENTER (BIT2)
     + /A4*    /A2* A1* A0            ; INCREMENTER (BIT2)
     +  A4*/A3* A2*/A1                ; INCREMENTER (BIT2)
     +  A4*/A3*/A2* A1* A0             ; INCREMENTER (BIT2)

NA3  = /A4* A3*/A2                    ; INCREMENTER (BIT3)
     + /A4* A3*     /A1               ; INCREMENTER (BIT3)
     + /A4* A3*          /A0          ; INCREMENTER (BIT3)
     + /A4*/A3* A2* A1* A0             ; INCREMENTER (BIT3)

NA4  = /A4* A3* A2* A1* A0            ; INCREMENTER (MSB)
     +  A4*/A3*/A2                    ; INCREMENTER (MSB)
     +  A4*/A3*     /A1               ; INCREMENTER (MSB)

; TIMING WAVEFORMS

TVCC = /A4                            ; TIMING FOR VCC
     +  A4*/A3*/A2*/A1
     +  A4*/A3*/A2*     /A0

TP01 = /A4*/A3* A2                    ; TIMING FOR PIN 01
     + /A4*/A3*      A1
     + /A4*/A3*           A0
     + /A4* A3*/A2*/A1*/A0

TP11 = /A4* A3* A2                    ; TIMING FOR PIN 11
     + /A4* A3*      A1
     +  A4*/A3*/A2*/A1
```

4

FUNCTION TABLE

A4 A3 A2 A1 A0 NA4 NA3 NA2 NA1 NA0 TVCC TP01 TP11

;	NNNNN						
;AAAAA	AAAAA	TIMING	WAVEFORMS				
;43210	43210	TVCCP	TP01	TP11	; ## ;	COMMENTS	
LLLLL	LLLLH	H	L	L	; 01 ;	ASSERT TVCC, START HERE	
LLLLH	LLLHL	H	H	L	; 02 ;	ASSERT TP01	
LLLHL	LLLHH	H	H	L	; 03 ;		
LLLHH	LLHLL	H	H	L	; 04 ;		
LLHLL	LLHLH	H	H	L	; 05 ;		
LLHLH	LLHHL	H	H	L	; 06 ;		
LLHHL	LLHHH	H	H	L	; 07 ;		
LLHHH	LHLLL	H	H	L	; 08 ;		
LHLLL	LHLLH	H	H	L	; 09 ;	CLEAR TP01	
LHLLH	LHLHL	H	L	L	; 10 ;	ASSERT TP11	
LHLHL	LHLHH	H	L	H	; 11 ;		
LHLHH	LHHLL	H	L	H	; 12 ;		
LHHLL	LHHLH	H	L	H	; 13 ;		
LHHLH	LHHHL	H	L	H	; 14 ;		
LHHHL	LHHHH	H	L	H	; 15 ;		
LHHHH	HLLLL	H	L	H	; 16 ;		
HLLLL	HLLLH	H	L	H	; 17 ;		
HLLLH	HLLHL	H	L	H	; 18 ;		
HLLHL	HLLHH	H	L	L	; 19 ;	CLEAR TP11	
HLLHH	HLHLL	L	L	L	; 20 ;	CLEAR TVCC	
HLHLL	HLHLH	L	L	L	; 21 ;		
HLHLH	HLHLH	L	L	L	; 22 ;	LOOP HERE UNTIL RESET	

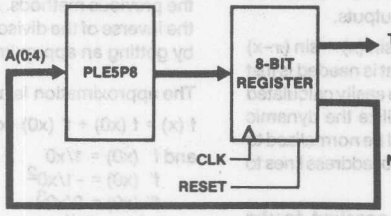
TIMING GENERATOR FOR PAL PROGRAMMING (cont'd)

DESCRIPTION

THIS LOGIC SPECIFICATION IS A TIMING SIGNAL GENERATOR TO BE USED FOR SECURITY FUSE PROGRAMMING OF PAL DEVICES. A PLE5P8 FOLLOWED BY AN 8-BIT REGISTER ARE USED TO IMPLEMENT THIS FUNCTION.

THE PLE CONTAINS TWO FUNCTIONS IN THE SINGLE CHIP. THE FIRST FUNCTION IS A UNIQUE COUNTER USED FOR NEXT ADDRESS GENERATION. THE COUNTER INCREMENTS UP TO COUNT-21 AND THEN LOCKS UP THE INCREMENTAL OPERATION AT COUNT-22. THE SECOND FUNCTION IS A TIMING GENERATOR USED FOR DEFINING TIMING RELATIONSHIP AMOUNG VCC, P01, AND P11 SIGNALS (SEE "PAL PROGRAMMING" IN THE PAL DATA SHEET FOR MORE DETAILS.)

THE SCHEMATIC IS AS FOLLOWS:



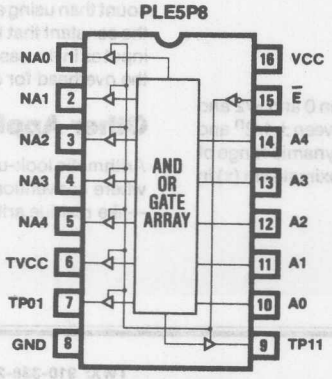
THIS LOGIC OUTPUTS A SEQUENCE OF TIMING PATTERNS DURING THE INCREMENTAL OPERATION AND THEN HOLDS ALL OUTPUTS LOW UNTIL RESET SIGNAL FOR THE 8-BIT REGISTERED IS APPLIED.

APPLYING 200 KHz CLOCK SIGNAL TO THE CLK INPUT OF THE REGISTER, THE FOLLOWING TIMINGS ARE GENERATED:

1. VCC WIDTH : 95 usec
2. TPP : 40 usec
3. tD : 5 usec

BY APPLYING THIS DESIGN METHOD, WE CAN EASILY GENERATE A SEQUENCE OF UNIQUELY DEFINED PATTERNS EACH TIME THE RESET PULSE IS APPLIED.

TIMING GENERATOR FOR PAL SECURITY FUSE PROGRAMMING



Fast Arithmetic Look-up

Fast Arithmetic Look-up

In performing arithmetic operations like trigonometric functions, multiplications and division, in order to reduce the delay, look-up tables are often used.

Sine Look-up

For trigonometric functions like sine function, it is very time-consuming to generate the function using the polynomial which represents the function. PLEs can provide a very good alternative for sine look-up. An example is to use a 2 K x 8 PLE to do a sine look-up of an 11-bit input to 8-bit sine outputs.

Since sine function has the following property: $\sin(x) = \sin(\pi - x) = -\sin(\pi + x) = -\sin(2\pi - x) = \sin(2\pi + x)$, what is needed is just the sine function for $0 < x < \pi/2$, the rest can be easily calculated using the above relations. In order to fully utilize the dynamic range, the inputs of the sine look-up PLE should be normalized to $(\pi/2) / (2^n) = \pi/[2^{n+1}]$ where n is the number of address lines to the PLE.

Since n is fixed for the PLE chosen, and π is a constant, for the look-up table $\pi/[2^{n+1}]$ is a constant. Therefore, if the sine function of a given x is to be found, x will first be multiplied by the constant $[2^{n+1}]/\pi$ and sent to the address of the PLE to get the final result.

Cos (x) is related to sine function as $\sin(\pi/2 - x)$. Thus cosine function can also be found in the same manner by using $\pi/2 - x$ instead of just x. Other functions like tangent, secant etc., can also be found as a function of sine.

To increase the dynamic range of outputs, we can just use another PLE, say, 2 K x 8 PLE, to generate the less-significant bits of the sine function.

If a larger dynamic range is needed for the inputs, result may be approximated using the Taylor series:

$$f(x) = f(x0) + f'(x0)(x-x0) + 1/2 f''(x0)(x-x0)^2 + \ldots$$

where f' and f'' are the first and second derivations of f. Since x0 by itself represents a dynamic range of 2^{-n}, and x is x0 concatenated with the rest of the bits, x–x0 must lie between 0 and $1/2^n$. For f (x) = sin (x),

$$f(x0) = \sin(x0)$$
$$f'(x0) = \cos(x0)$$
$$\text{and } f''(x0) = -\sin(x0)$$

So f'' (x0) is between –1 and 0 for x0 lies between 0 and $\pi/2$ and $|x-x0| < 2^{-n}$. Therefore, the last term will be between $\pm 1/2^n$ and 0 and as long as we do not want to expand the dynamic range of x beyond 2n–bits, it should be sufficient to approximate sin (x) in the first two terms:

$$\sin(x) = \sin(x0) + \cos(x0)(x-x0)$$

Since x–x0 is represented by only the bits after the more significant n-bits, and $\cos(x0) = \sin(\pi/2 - x0)$, the implementation will be very simple.

Division

Division will normally be much slower than multiplication. There are several ways to perform division. Bit-by-bit division restoring and nonrestoring algorithms are generally very slow. Another way is to use several bits at a time division which is faster than the previous methods. A third way is to multiply the dividend by the inverse of the divisor. The inverse of the divisor can be found by getting an approximation followed by iterations.

The approximation is again given by the Taylor series:

$$f(x) = f(x0) + f'(x0)(x-x0) + 1/2 f''(x0)(x-x0)^2 + \ldots$$

$$\text{and } f(x0) = 1/x0$$
$$f'(x0) = -1/x0^2$$
$$f''(x0) = 2/x0^3$$

Say x0 is 8-bit long and the first approximation of the inverse is found using a 256 x 8 PLE. The first approximation can be obtained by subtracting $(x-x0)/(x0^2)$. Since the first approximation is limited by an error of approximately $(x-x0)^3/x0^3$ and if x0 is at least 1, the error is limited by approximately $(x-x0)^2$. Since x0 has a 8-bit resolution, and x–x0 is represented by the rest of the bits. The resolution of the second approximation will be about 16 bits. The third approximation is similarly deduced and has a resolution of about 32 bits, and the fourth has a resolution of about 64 bits.

The inverse thus obtained is then multiplied by the dividend to give the quotient.

Scaling

In arithmetic operations, scaling is sometimes needed. Scaling normally involves multiplication or division by a constant. If this constant can be expressed in 2^n where n is an integer, then scaling is simply shifting. Scaling with other constants may need a multiplier. A multiplier is more expensive and has a higher pin count than using a PLE. A PLE can reduce the pin count because the constant that the operand is to be scaled is not required as an input as in the case of a multiplier. This will tremendously reduce the overhead for data scaling.

Other Applications

Arithmetic look-up are also very useful for arithmetic operations where conventional binary integral arithmetic is not applicable —like residue arithmetic, and distributed arithmetic.

```
PLE8P8                          PLE DESIGN SPECIFICATION
P5018                             VINCENT COLI 12/08/82
4-BIT MULTIPLIER LOOK-UP TABLE
MMI SANTA CLARA, CALIFORNIA
.ADD X0 X1 X2 X3 Y0 Y1 Y2 Y3
.DAT S0 S1 S2 S3 S4 S5 S6 S7


S7,S6,S5,S4,S3,S2,S1,S0 = X3,X2,X1,X0 .*. Y3,Y2,Y1,Y0     ; S = X * Y

FUNCTION TABLE

X3 X2 X1 X0 Y3 Y2 Y1 Y0 S7 S6 S5 S4 S3 S2 S1 S0

;-OPERANDS-     PRODUCTS
;XXXX  YYYY     SSSSSSSS      COMMENTS
;3210  3210     76543210
-----------------------------------------
 LLLL  LLLL     LLLLLLLL      0 *  0  =   0
 LLLH  HHHH     LLLLHHHH      1 * 15  =  15
 HHHH  LLLH     LLLLHHHH     15 *  1  =  15
 HHHH  HHHH     HHHLLLLH     15 * 15  = 225
-----------------------------------------
```

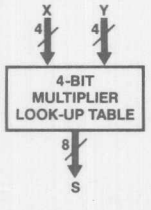
4

DESCRIPTION

THIS PLE8P8 PERFORMS 4-BIT LOOK-UP TABLE MULTIPLICATION. THE DEVICE ACCEPTS TWO 4-BIT OPERANDS (X3-X0 AND Y3-Y0) TO PRODUCE THE 8-BIT PRODUCT (S7-S0). THE PLE8P8 ALSO HAS THREE-STATE OUTPUTS WITH TWO ACTIVE-LOW OUTPUT ENABLE CONTROL PINS (/E1 AND /E2).

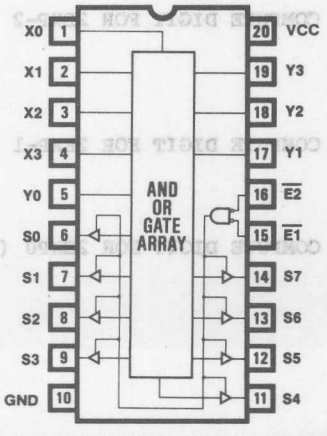
```
            X3 X2 X1 X0 }
        X   Y3 Y2 Y1 Y0 } TWO 4-BIT OPERANDS
-----------------------
S7 S6 S5 S4 S3 S2 S1 S0
```

8-BIT PRODUCT



4-BIT MULTIPLIER LOOK-UP TABLE

PLE8P8



```
P5023                                          PETER ZECHERLE 03/06/84
ARC TANGENT LOOK-UP TABLE
MMI GMBH MUNICH
.ADD A0 A1 A2 A3 A4
.DAT F0 F1 F2 F3 F4 F5 F6 F7


F0 =     A1*     /A3*/A4    ; COMPUTE DIGIT FOR 2EXP-7 (0.00078125) (LSB)
   +          A2*/A3
   +  A0*/A1*     A3*/A4
   + /A0* A1*        /A4
   + /A0*/A1*    /A3* A4
   +  A0*     A2
   +      A1* A2

F1 =     /A1*     A3*/A4      ; COMPUTE DIGIT FOR 2EXP-6 (0.015625)
   +  A0*         /A3* A4
   +      A1*    /A3* A4
   +          A2*/A3* A4
   +  A0* A1*    /A3
   +  A0*     A2*/A3
   + /A0* A1* A2*    /A4
   +  A0*    /A2* A3*/A4

F2 =  A0*         /A3*/A4      ; COMPUTE DIGIT FOR 2EXP-5 (0.03125)
   +      A1*/A2*    /A4
   +                A3* A4
   + /A0*      A2* A3*/A4
   +     /A1* A2* A3*/A4

F3 =      A1*/A2*    /A4      ; COMPUTE DIGIT FOR 2EXP-4 (0.0625)
   +     /A1* A2*    /A4
   + /A0*           A3*/A4
   +     /A1*       A3*/A4

F4 =     /A1*      A3*/A4     ; COMPUTE DIGIT FOR 2EXP-3 (0.125)
   +  A0* A1*/A2*    /A4
   +      A1* A2*/A3*/A4
   + /A0*           A3*/A4

F5 =  A0*/A1*         /A4      ; COMPUTE DIGIT FOR 2EXP-2 (0.25)
   +          A2*/A3*/A4
   +         /A2* A3*/A4
   + /A0*           A3*/A4

F6 =  A0*/A1*/A2*/A3           ; COMPUTE DIGIT FOR 2EXP-1 (0.5)
   +  A0* A1* A2* A3
   +                   A4

F7 =      A1                   ; COMPUTE DIGIT FOR 2EXP0 (1) (MSB)
   +          A2
   +              A3
   +                  A4
```

ARC TANGENT LOOK-UP TABLE (cont'd)

FUNCTION TABLE

;----ANGLE----					--------F = ARCTAN(A)--------									
; INTEGER					INTEGER	FRACTIONS							---F = ARCTAN(A)---	
A4	A3	A2	A1	A0	F7	F6	F5	F4	F3	F2	F1	F0	;ANGLE	LOOK-UP	CALCULATED
L	L	L	L	L	L	L	L	L	L	L	L	L	0	0.0000	0.0000
L	L	L	L	H	L	H	H	L	L	H	L	L	1	0.7813	0.7854
L	L	L	H	L	H	L	L	L	H	H	L	H	2	1.1016	1.1071
L	L	H	L	L	H	L	H	L	H	L	L	H	4	1.3203	1.3258
L	L	H	L	H	H	L	H	L	H	H	H	H	5	1.3672	1.3734
L	H	L	L	L	H	L	H	H	H	L	H	L	8	1.4531	1.4464
H	L	L	L	L	H	H	L	L	L	L	L	H	16	1.5078	1.5084
H	H	H	H	H	H	H	L	L	L	H	L	H	31	1.5391	1.5385

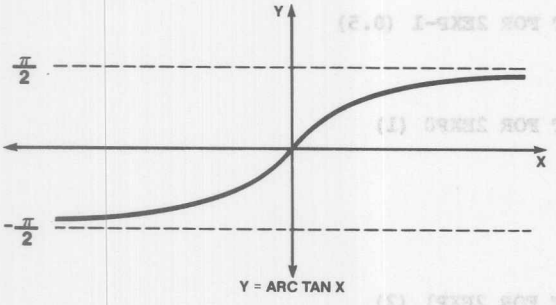
DESCRIPTION

THIS APPLICATION ILLUSTRATES THE CALCULATION OF THE ARC TANGENT FUNCTION USING A PLE5P8 AS A LOOK-UP TABLE. OTHER TRIGONOMETRIC FUNCTIONS (SUCH AS SINE, COSINE, COTANGENT, SECANT, COSECANT AND THEIR ARC INVERSE EQUIVALENT FUNCTIONS) OR HYPERBOLIC FUNCTIONS CAN ALSO BE CONSTRUCTED USING PLES AS LOOK-UP TABLES.

F = ARCTAN(A) WHERE F = ARC TANGENT OF A
A = ANGLE IN RADIANS

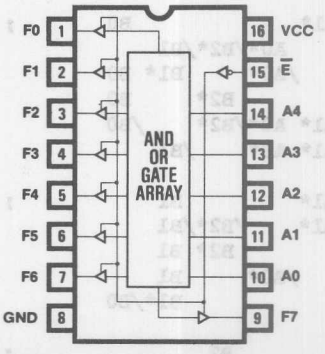
EXAMPLE: FOR A = 5, F = ARCTAN(5) = 1.3672

A PLE WITH MORE INPUTS, SUCH AS THE PLE11P8, SHOULD BE USED TO CONSTRUCT A LOOK-UP TABLE WHEN ADDITIONAL ACCURACY IS REQUIRED.



ARC TANGENT LOOK-UP TABLE

PLE5P8



4

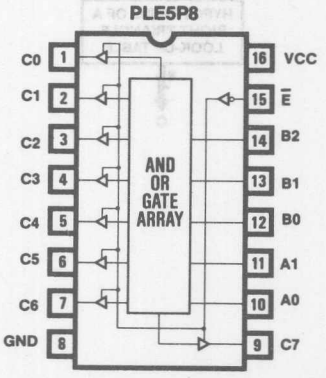
```
PLE5P8                                          PLE DESIGN SPECIFICATION
P5024                                             WILLY VOLDAN 06/02/84
HYPOTENUSE OF A RIGHT TRIANGLE LOOK-UP TABLE
MMI GMBH MUNICH
.ADD A0 A1 B0 B1 B2
.DAT C0 C1 C2 C3 C4 C5 C6 C7

C0 =      A0*/B2* B1        ; COMPUTE DIGIT FOR 2EXP-5 (0.03125) (LSB)
   + /A1* A0* B2*/B1
   +  A1*    /B2*     B0
   +  A1*    /B2* B1
   +  A1*/A0* B2*/B1*/B0
   +  A1* A0*    B1* B0
   +     A0*/B2*     B0

C1 =      A0*     B1*/B0    ; COMPUTE DIGIT FOR 2EXP-4 (0.0625)
   + /A1* A0* B2
   +  A1*/A0*/B2*     B0
   +  A1*/A0* B2*    /B0
   +     A0* B2*     B0
   +  A1* A0*    B1

C2 =      A0*/B2*     B0    ; COMPUTE DIGIT FOR 2EXP-3 (0.125)
   + /A1* A0*/B2* B1
   +  A1*/A0* B2*/B1
   +  A1* A0* B2* B1
   +  A1*    /B2*/B1* B0

C3 = /A1* A0*/B2*/B1* B0    ; COMPUTE DIGIT FOR 2EXP-2 (0.25)
   +  A1*/A0*     B1*/B0
   +  A1*/A0* B2
   +  A1*     B2*     B0

C4 =  A1*/A0*/B2* B1        ; COMPUTE DIGIT FOR 2EXP-1 (0.5)
   +  A1* A0* B2*/B1* B0
   +  A1* A0*     B1*/B0

C5 = /A1*             B0    ; COMPUTE DIGIT FOR 2EXP0 (1)
   +     A0*/B2*/B1
   +    /A0*    B1* B0
   +        B2*     B0
   +  A1* A0*/B2*    /B0
   +  A1* A0*    /B1

C6 = /A1*             B1    ; COMPUTE DIGIT FOR 2EXP1 (2)
   +  A1*    /B2*/B1
   +        B2* B1
   +    /A0*     B1
   +             B1*/B0

C7 =          B2            ; COMPUTE DIGIT FOR 2EXP2 (4) (MSB)
   +  A1* A0*     B1* B0
```

HYPOTENUSE OF A RIGHT TRIANGLE LOOK-UP TABLE (cont'd)

FUNCTION TABLE

;-LENGTH OF SIDES-							LENGTH OF THE HYPOTENUSE									
;SIDE A			SIDE B				INTEGER			FRACTION					SIDES	LENGTH OF HYPOTENUSE
A1	A0		B2	B1	B0		C7	C6	C5	C4	C3	C2	C1	C0	;A B	LOOK-UP	CALCULATED
L	L		L	L	L		L	L	L	L	L	L	L	L	0 0	0.00	0.00
L	L		L	L	H		L	L	H	L	L	L	L	L	0 1	1.00	1.00
L	L		L	H	L		L	H	L	L	L	L	L	L	0 2	2.00	2.00
L	L		H	L	L		H	L	L	L	L	L	L	L	0 4	4.00	4.00
L	H		L	L	L		L	L	H	L	L	L	L	L	1 0	1.00	1.00
H	L		L	L	L		L	H	L	L	L	L	L	L	2 0	1.00	1.00
H	L		L	H	L		L	H	L	H	H	L	L	H	2 2	2.78	2.83
H	L		H	L	L		H	L	L	L	H	H	H	H	2 4	4.47	4.47
H	H		L	H	L		L	H	H	H	L	L	H	H	3 2	3.59	3.61
H	H		H	H	H		H	H	H	L	H	H	H	H	3 7	7.47	7.62

HYPOTENUSE OF A RIGHT TRIANGLE LOOK-UP TABLE



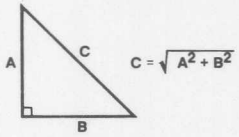
DESCRIPTION

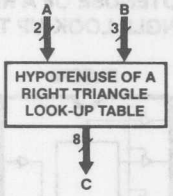
THE GENERATION OF COMPLEX ARITHMETIC FUNCTIONS SUCH AS THE PATHAGORAS THEOREM IS USUALLY VERY DIFFICULT TO IMPLEMENT DIRECTLY IN HARDWARE. HOWEVER, IMPLEMENTING THE FUNCTION AS A LOOK-UP TABLE USING A PLE GREATLY SIMPLIFIES THE PROBLEM.

THIS EXAMPLE ILLUSTRATES HOW TO IMPLEMENT A LOOK-UP TABLE IN A PLE5P8 WHICH CALCULATES THE LENGTH OF THE HYPOTENUSE OF A RIGHT TRIANGLE AS A FUNCTION OF THE LENGTH OF THE TWO REMAINING SIDES OF THE TRIANGLE. THE THEOREM OF PATHAGORAS STATES THAT THE LENGTH OF THE HYPOTENUSE OF A RIGHT TRIANGLE IS EQUAL TO THE SQUARE ROOT OF THE SUM OF THE SQUARE OF THE OTHER TWO SIDES OR C = SQRT(A**2 + B**2). THE INPUTS, "A" AND "B", CORRESPOND TO THE SIDES ADJACENT TO THE RIGHT ANGLE (I.E. 90 DEGREE ANGLE), WHILE THE OUTPUT, "C", CORRESPONDS TO THE SIDE OPPOSITE TO THE RIGHT ANGLE WHICH IS CALLED THE HYPOTENUSE.

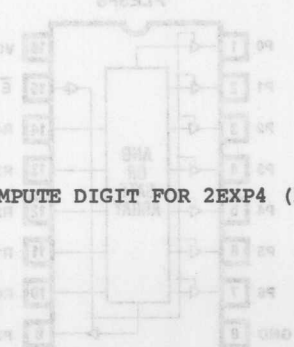
C = SQRT(A**2 + B**2) WHERE C = LENGTH OF SIDE C (THE HYPOTENUSE)
A = LENGTH OF SIDE A
B = LENGTH OF SIDE B

EXAMPLE: FOR A = 2 AND B = 4, C = SQRT(2**2 + 4**2) = 4.47





```
PLE5P8                                          PLE DESIGN SPECIFICATION
P5025                                           PETER WITTFOTH 06/02/84
PERIMETER OF A CIRCLE LOOK-UP TABLE
MMI GMBH MUNICH
.ADD R0 R1 R2 R3 R4
.DAT P0 P1 P2 P3 P4 P5 P6 P7


P0 =     /R1* R2*/R3*/R4      ; COMPUTE DIGIT FOR 2EXP0 (1) (LSB)
   + /R0*/R1* R2*    /R4
   +      R1* R2*     R4
   +      R1*/R2*/R3
   +  R0*/R1*/R2* R3
   + /R0* R1*/R2
   +      R1*/R2*    /R4
   +     /R1*/R2*     R4

P1 =  R0*    /R2*/R3      ; COMPUTE DIGIT FOR 2EXP1 (2)
   + /R0* R1* R2*/R3
   + /R0*    /R2* R3
   +  R0*     R2* R3
   + /R0*     R2*/R3* R4
   +  R0*/R1* R2*    /R4
   +     /R1* R2* R3*/R4
   +  R0* R1*    R3* R4

P2 =  R0*/R1*    /R3*/R4      ; COMPUTE DIGIT FOR 2EXP2 (4)
   +  R0* R1*/R2*/R3* R4
   + /R0* R1* R2* R3* R4
   + /R0* R1*/R2*   /R4
   +      R1* R2*/R3*/R4
   +  R0* R1*    R3*/R4
   + /R0*/R1*/R2*    R4
   +     /R1* R2*/R3* R4
   +  R0*/R1*   R3* R4

P3 = /R0* R1*/R2*    /R4      ; COMPUTE DIGIT FOR 2EXP3 (8)
   +  R0*/R1*/R2* R3
   +  R0*/R1*/R2*     R4
   +  R0*    /R2* R3* R4
   +  R0*     R2*/R3*/R4
   + /R0*     R2* R3*/R4
   +      R1* R2* R3*/R4
   + /R0*     R2*/R3* R4
   +      R1* R2*/R3* R4
   + /R0* R1* R2*     R4
   + /R0*/R1* R2*/R3

P4 =  R0* R1*/R2*/R3          ; COMPUTE DIGIT FOR 2EXP4 (16)
   +      R1*/R2*/R3* R4
   + /R0*/R1* R2*/R3
   +  R0*/R1* R2*    /R4
   +     /R1*/R2* R3
   + /R0* R1*     R3*/R4
   +      R1* R2* R3*/R4
   +     /R1*     R3* R4
   + /R0*     R2* R3* R4
```



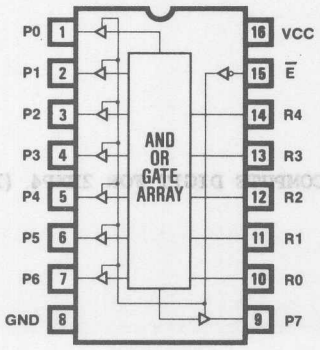
PERIMETER OF A CIRCLE LOOK-UP TABLE (cont'd)

```
P5 =      R1* R2*/R3*/R4      ; COMPUTE DIGIT FOR 2EXP5 (32)
   +     /R1*/R2* R3*/R4
   +        /R2*/R3* R4
   +      R1*/R2*     R4
   +     /R1* R2* R3* R4
   + /R0* R1*/R2* R3
   + /R0*/R1* R2*     R4
   + /R0*     R2* R3* R4

P6 =  R0* R1*     R3*/R4      ; COMPUTE DIGIT FOR 2EXP6 (64)
   + /R0*/R1*    /R3* R4
   +          R2* R3*/R4
   +         /R2*/R3* R4
   +  R0* R1* R2* R3

P7 =  R0*     R2*     R4      ; COMPUTE DIGIT FOR 2EXP7 (128) (MSB)
   +     R1* R2*     R4
   +             R3* R4

FUNCTION TABLE
```

;---RADIUS----					-------PERIMETER-------										
; INTEGER					MSB		INTEGER					LSB		PERIMETER OF A CIRCLE	
R4	R3	R2	R1	R0	P7	P6	P5	P4	P3	P2	P1	P0	;RADIUS	LOOK-UP	CALCULATED
L	L	L	L	L	L	L	L	L	L	L	L	L	0	0	0.0
L	L	L	L	H	L	L	L	L	L	L	H	L	1	6	6.3
L	L	L	H	L	L	L	L	L	H	H	L	H	2	13	12.6
L	L	L	H	H	L	L	L	H	L	L	H	H	3	19	18.8
L	L	H	L	L	L	L	L	H	H	L	L	H	4	25	25.1
L	H	L	L	L	L	L	H	H	L	L	H	L	8	50	50.3
H	L	L	L	L	L	H	H	L	L	H	L	H	16	101	100.5
H	H	H	H	H	H	H	L	L	L	L	H	H	31	195	194.8

PERIMETER OF A CIRCLE LOOK-UP TABLE

PLE5P8



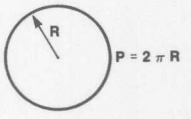
PERIMETER OF A CIRCLE LOOK-UP TABLE (cont'd)

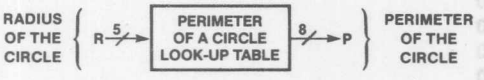
DESCRIPTION

THIS EXAMPLE ILLUSTRATES HOW TO IMPLEMENT A LOOK-UP TABLE IN A PLE5P8 FOR THE PERIMETER OF A CIRCLE AS A FUNCTION OF THE RADIUS. THE INPUT PINS (R4-R0), WHICH REPRESENT THE RADIUS OF A CIRCLE, ARE MULTIPLIED BY 2 TIMES PI IN ORDER TO CALCULATE THE PERIMETER OF A CIRCLE (P7-P0). THIS LOOK-UP TABLE IS VALID FOR RADII BETWEEN 0 AND 31. A PLE8P8 SHOULD BE USED INSTEAD IF A LARGER RADIUS RANGE (BETWEEN 0 AND 81) IS REQUIRED.

P = 2*PI*R WHERE P = PERIMETER OF THE CIRCLE
PI = 3.1415
R = RADIUS OF THE CIRCLE (BETWEEN 0 AND 31)

EXAMPLE: FOR R = 3, P = 2*PI*3 = 19





4

```
MMI GMBH MUNICH
.ADD L0 L1 L2 L3 L4
.DAT T0 T1 T2 T3 T4 T5 T6 T7


T0 = /L4*     /L2*/L1* L0          ; COMPUTE DIGIT FOR 2EXP-5 (0.03125) (LSB)
   +      /L3*/L2* L1*/L0
   +       L3*/L2*     L0
   + /L4*      L2*/L1*/L0
   +  L4* L3*          L0
   +  L4*/L3*      L1
   +  L4*/L3* L2*     /L0

T1 =          /L2* L1*/L0          ; COMPUTE DIGIT FOR 2EXP-4 (0.0625)
   + /L4*/L3* L2*      L0
   + /L4* L3*/L2* L1
   + /L4*      L2*/L1*/L0
   +  L4*/L3*/L2* L1
   +      /L3* L2*/L1
   +  L4* L3*     /L1* L0
   +  L4* L3*      L1*/L0

T2 = /L4*/L3*      L1*/L0          ; COMPUTE DIGIT FOR 2EXP-3 (0.125)
   + /L4* L3*/L2*      L0
   +  L4*/L3*/L2*/L1*/L0
   +  L4*/L3*      L1* L0
   +  L4* L3*/L2* L1*/L0
   +  L4* L3* L2*/L1
   +  L4*      L2*      L0
   + /L4*/L3*     /L1* L0
   + /L4*/L3* L2*     /L0

T3 = /L4* L3*      L1* L0          ; COMPUTE DIGIT FOR 2EXP-2 (0.25)
   +  L4*/L3*      L1
   +  L4* L3* L2*/L1
   +      /L3*/L2*/L1* L0
   +      /L3* L2* L1* L0
   +       L3*/L2* L1* L0
   +       L3* L2*/L1* L0
   + /L4*      L2*/L1* L0

T4 = /L4*/L3*      L1*/L0          ; COMPUTE DIGIT FOR 2EXP-1 (0.5)
   +      /L3* L2* L1
   + /L4* L3* L2*/L1
   +  L4*/L3* L2
   +  L4*      L2* L1
   +           L2* L1*/L0
```

PERIOD OF OSCILLATION FOR A PENDULUM LOOK-UP TABLE (cont'd)

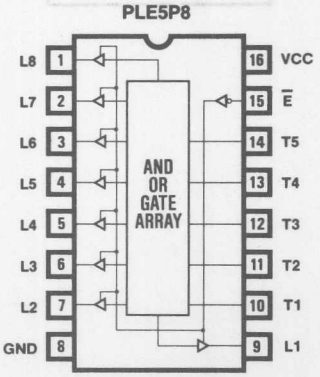
```
T5 = /L4*    /L2*    /L0          ; COMPUTE DIGIT FOR 2EXP0 (1)
   + /L4*    /L2*/L1
   +      L3*     /L0
   +      L3*/L2
   +      L3*    /L1
   +  L4* L3

T6 = /L4*    /L2* L1* L0          ; COMPUTE DIGIT FOR 2EXP1 (2)
   + /L4* L3*       /L0
   + /L4*/L3* L2
   + /L4*/L3*    /L1

T7 =      L3* L2* L1* L0          ; COMPUTE DIGIT FOR 2EXP2 (4) (MSB)
   +  L4
```

FUNCTION TABLE

;--AMPLITUDE-- INTEGER					--PERIOD OF OSCILLATION-- INTEGER				FRACTION					PERIOD OF OSCILLATION	
L4	L3	L2	L1	L0	T7	T6	T5	T4	T3	T2	T1	T0	;AMPLITUDE	LOOK-UP	CALCULATED
L	L	L	L	L	L	L	H	L	L	L	L	L	1	2.0000	2.0050
L	L	L	L	H	L	L	H	L	H	H	L	H	2	2.8125	2.8356
L	L	L	H	L	L	L	H	H	L	H	H	H	3	3.4375	3.4728
L	L	H	L	L	L	H	L	L	L	H	H	H	5	4.4375	4.4834
L	H	L	L	L	L	H	H	L	L	L	L	L	9	6.0000	6.0151
H	L	L	L	L	H	L	L	L	L	H	L	L	17	8.2500	8.2670
H	H	H	H	H	H	L	H	H	L	H	L	H	32	11.3125	11.3423

PERIOD OF OSCILLATION FOR A MATHEMATICAL PENDULUM LOOK-UP TABLE

PLE5P8



PERIOD OF OSCILLATION FOR A PENDULUM LOOK-UP TABLE (cont'd)

DESCRIPTION

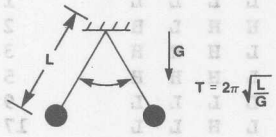
THIS PLE5P8 IS USED TO IMPLEMENT A LOOK-UP TABLE FOR THE PERIOD OF OSCILLATION OF A MATHEMATICAL PENDULUM. THE PERIOD OF OSCILLATION FOR MATHEMATICAL PENDULUM (T) IS DEPENDENT UPON ITS AMPLITUDE OF SWING (L) AND THE ACCELERATION DUE TO GRAVITY (G). THE PERIOD OF OSCILLATION IS CALCULATED USING THE FOLLOWING EQUATION:

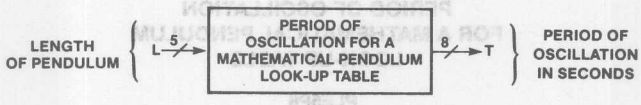
T = 2*PI*SQRT(L/G) WHERE T = PERIOD OF OSCILLATION IN SECONDS
PI = 3.14
L = AMPLITUDE OF SWING IN METERS
G = ACCELERATION DUE TO GRAVITY IN M/S/S (9.81 M/S/S)

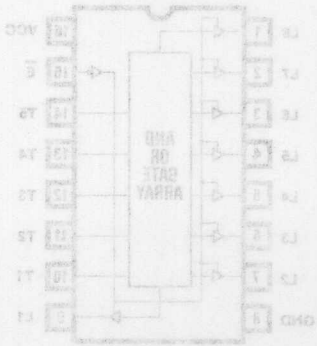
EXAMPLE: FOR L = 5, T = 2*PI*SQRT(5/G) = 4.4375

A PLE WITH 5 INPUTS CAN BE USED TO CALCULATE THE PERIOD OF OSCILLATION FOR AMPLITUDES UP TO L = 32 METERS. PLES WITH MORE INPUTS SHOULD BE USED TO CALCULATE LARGER PERIODS OF OSCILLATION.

THIS EXAMPLE DEMONSTRATES HOW EASY IT IS TO CONSTRUCT LOOK-UP TABLES FOR COMPLEX ARITHMETIC FUNCTIONS USING PLES.



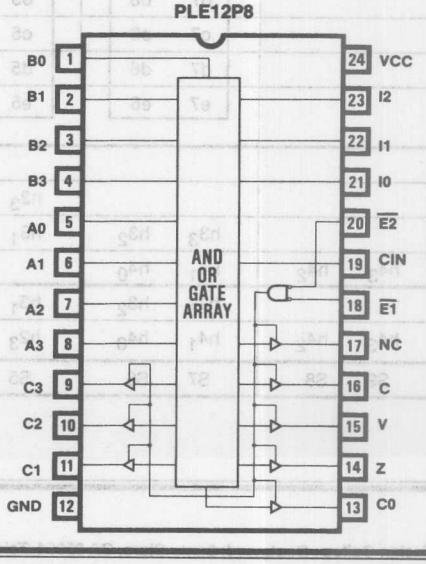




```
PLE12P8                                          PLE DESIGN SPECIFICATON
P5017                                               FRANK LEE 10/14/83
ARITHMETIC LOGIC UNIT
MMI SANTA CLARA, CALIFORNIA
.ADD A3 A2 A1 A0 B3 B2 B1 B0 CIN I2 I1 I0
.DAT C3 C2 C1 C0 Z V C

;************************************************
;*  THIS DESIGN IS NOT YET SUPPORTED BY PLEASM  *
;************************************************

C,C3,C2,C1,C0 = /S2*/S1* S0*/A3,/A2,/A1,/A0          ;B - A - 1 + CIN
                    .+. B3, B2, B1, B0.+. CIN
              + /S2* S1*/S0* A3, A2, A1, A0          ;A - B - 1 + CIN
                    .+./B3,/B2,/B1,/B0.+. CIN
              + /S2* S1* S0* A3, A2, A1, A0          ;A + B + CIN
                    .+. B3, B2, B1, B0.+. CIN
              +  S2*/S1*/S0*/A3,/A2,/A1,/A0          ;A XOR B
                    :*: B3, B2, B1, B0
              +  S2*/S1* S0* A3, A2, A1, A0          ;A + B
                    +  S2*/S1* S0* B3, B2, B1, B0
              +  S2* S1*/S0* A3, A2, A1, A0          ;A * B
                    * B3, B2, B1, B0
              +  S2* S1* S0                          ;PRESET

V             =  C:+: C3                             ;OVERFLOW

Z             = /C3*/C2*/C1*/C0                      ;ZERO

DESCRIPTION

THIS ALU CAN PERFORM 8 FUNCTIONS ON TWO 4-BIT OPERANDS A (A3-A0) AND
B (B3-B0) WITH CARRYIN (CIN) AND GIVES A 4-BIT RESULT C (C3-C0) WITH
CARRYOUT (C).  IT WILL ALSO GIVE STATUS AS OVERFLOW (V) AND ZERO (Z).

THE FUNCTION IS DETERMINED BY A 3-BIT FUNCTION SELECT CODE (S2-S0):
```

MODE	S2	S1	S0	FUNCTION
0	0	0	0	CLEAR
1	0	0	1	B - A - 1 + CIN
2	0	1	0	A - B - 1 + CIN
3	0	1	1	A + B + CIN
4	1	0	0	A XOR B
5	1	0	1	A + B
6	1	1	0	A * B
7	1	1	1	PRESET

ARITHMETIC LOGIC UNIT



Wallace Tree Compression

Wallace Tree Compression

In performing arithmetic calculations, it may happen that more than two numbers are to be added together. Adding two numbers can be achieved by using a simple adder. If there are more than two numbers to be summed, several levels of adders may be needed. This often causes too much delay.

An alternative is to use Wallace Tree Compression. Suppose there are m numbers each of n-bit wide. Summation over these numbers will range from 0 to m x (2^n-1) which will take $\log_2 [m (2^n-1) + 1]$ bits (rounded UP to the nearest integer). For example, if there are five 2-bit numbers, i.e., m = 5, and n = 2, the sum will be bounded by 5 x (2^2-1) = 15 which will need a total of 4 bits.

One Wallace Tree Compression by itself will not be very useful. But consider if five 8-bit integers are added together. This technique enables vertical compression of these numbers in four groups. This type of vertical compression also eliminates the need of carry propagation. The five numbers are represented by:

A = (a7, a6, a5, a4, a3, a2, a1, a0)
B = (b7, b6, b5, b4, b3, b2, b1, b0)
C = (c7, c6, c5, c4, c3, c2, c1, c0)
D = (d7, d6, d5, d4, d3, d2, d1, d0)
E = (e7, e6, e5, e4, e3, e2, e1, e0)

where the 7^{th} bits are the most significant, the calculation is as follows:

The groups are assigned as follows:

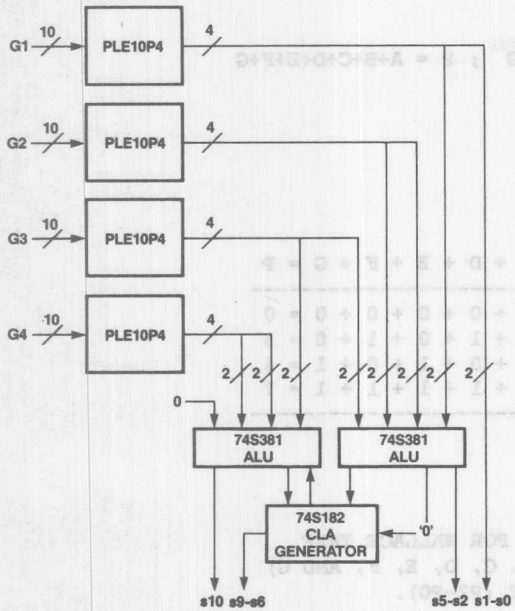
G1 : (a0, a1, b0, b1, c0, c1, d0, d1, e0, e1)
G2 : (a2, a3, b2, b3, c2, c3, d2, d3, e2, e3)
G3 : (a4, a5, b4, b5, c4, c5, d4, d5, e4, e5)
G4 : (a6, a7, b6, b7, c6, c7, d6, d7, e6, e7)

The above groups of bits can be compressed to:

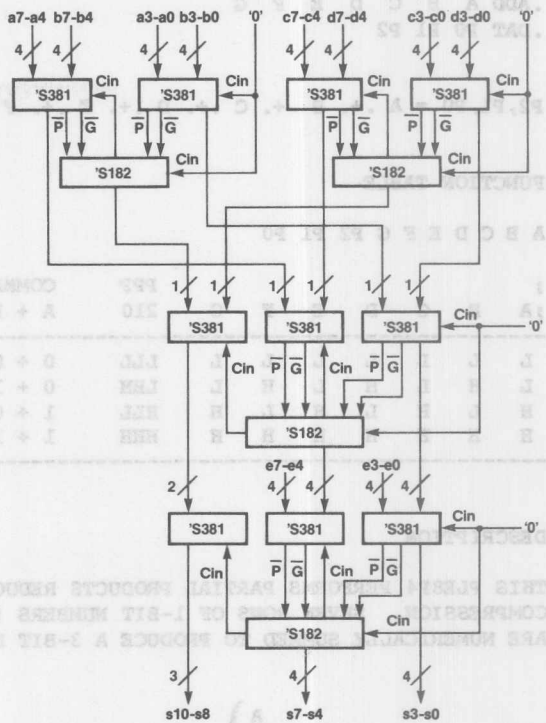
H1 : ($h1_3$, $h1_2$, $h1_1$, $h1_0$)
H2 : ($h2_3$, $h2_2$, $h2_1$, $h2_0$)
H3 : ($h3_3$, $h3_2$, $h3_1$, $h3_0$)
H4 : ($h4_3$, $h4_2$, $h4_1$, $h4_0$)

			G4		G3		G2		G1		
			a7	a6	a5	a4	a3	a2	a1	a0	= A
			b7	b6	b5	b4	b3	b2	b1	b0	= B
			c7	c6	c5	c4	c3	c2	c1	c0	= C
			d7	d6	d5	d4	d3	d2	d1	d0	= D
	+)		e7	e6	e5	e4	e3	e2	e1	e0	= E
							$h1_3$	$h1_2$	$h1_1$	$h1_0$	= H1
					$h2_3$	$h2_2$	$h2_1$	$h2_0$			= H2
			$h3_3$	$h3_2$	$h3_1$	$h3_0$					= H3
+)	$h4_3$	$h4_2$	$h4_1$	$h4_0$							= H4
			$h3_3$	$h3_2$	$h3_1$	$h3_0$	$h1_3$	$h1_2$	$h1_1$	$h1_0$	
+)	$h4_3$	$h4_2$	$h4_1$	$h4_0$	$h2_3$	$h2_2$	$h2_1$	$h2_0$			
S10	S9	S8	S7	S6	S5	S4	S3	S2	S1	S0	= result

addition of other bits. The hardware implementation is a follows:



alternative is using ten 74S381 ALUs and four 74S182 Carry Lookahead Generators.

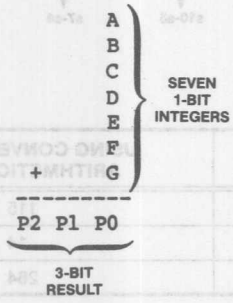


A comparison between the two architectures gives the following data:

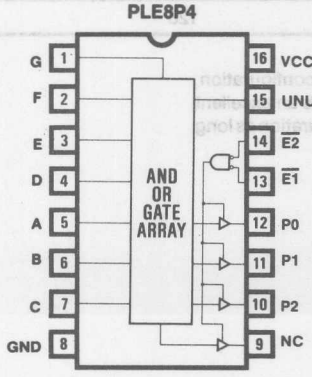
	USING WALLACE TREE COMPRESSION	USING CONVENTIONAL ARITHMETIC LOGIC
Delay (ns)	79	115
Number of components	7	14
Total number pins on the parts	128	264

Since Wallace Tree Compression can be of any configuration, there is no predefined part available. PLE provides an excellent solution. The designer may define his own configuration as long as it can be put in a commercially available PLE.

```
SEVEN 1-BIT INTEGER ROW PARTIAL PRODUCTS ADDER
MMI SANTA CLARA, CALIFORNIA
.ADD A  B  C  D  E  F  G
.DAT P0 P1 P2


P2,P1,P0 = A .+. B .+. C .+. D .+. E .+. F .+. G  ; P = A+B+C+D+E+F+G


FUNCTION TABLE

A B C D E F G P2 P1 P0

;                                   PPP     COMMENTS
;A   B   C   D   E   F   G          210     A + B + C + D + E + F + G = P
-------------------------------------------------------------------------
 L   L   L   L   L   L   L          LLL     0 + 0 + 0 + 0 + 0 + 0 + 0 = 0
 L   H   L   H   L   H   L          LHH     0 + 1 + 0 + 1 + 0 + 1 + 0 = 3
 H   L   H   L   H   L   H          HLL     1 + 0 + 1 + 0 + 1 + 0 + 1 = 4
 H   H   H   H   H   H   H          HHH     1 + 1 + 1 + 1 + 1 + 1 + 1 = 7
-------------------------------------------------------------------------


DESCRIPTION

THIS PLE8P4 PERFORMS PARTIAL PRODUCTS REDUCTION FOR WALLACE TREE
COMPRESSION.  SEVEN ROWS OF 1-BIT NUMBERS (A, B, C, D, E, F, AND G)
ARE NUMERICALLY SUMMED TO PRODUCE A 3-BIT RESULT (P2-P0).
```



SEVEN 1-BIT INTEGER ROW PARTIAL PRODUCTS ADDER

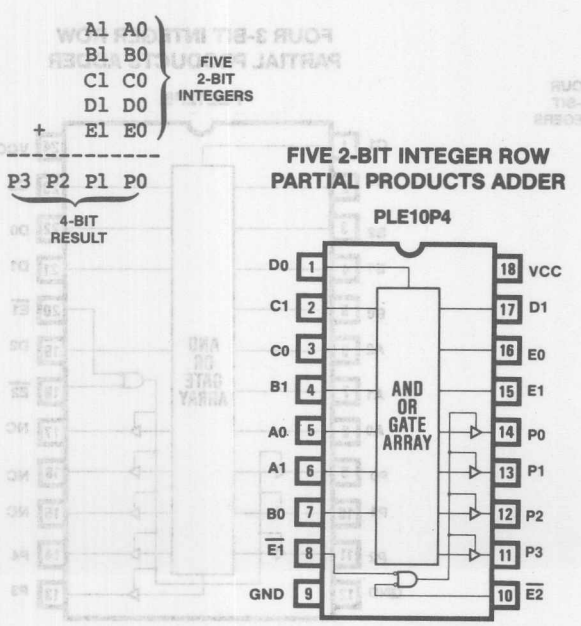


```
PLE10P4                                          PLE DESIGN SPECIFICATION
P5020                                              VINCENT COLI 08/22/83
FIVE 2-BIT INTEGER ROW PARTIAL PRODUCTS ADDER
MMI SANTA CLARA, CALIFORNIA
.ADD A0 A1 B0 B1 C0 C1 D0 D1 E0 E1
.DAT P0 P1 P2 P3


P3,P2,P1,P0 = A1,A0 .+. B1,B0 .+. C1,C0 .+. D1,D0 .+. E1,E0  ; P = A+B+C+D+E


FUNCTION TABLE

A1 A0 B1 B0 C1 C0 D1 D0 E1 E0 P3 P2 P1 P0

;AA   BB   CC   DD   EE    PPPP     COMMENTS
;10   10   10   10   10    3210     A + B + C + D + E = P
-------------------------------------------------------------
 LL   LL   LL   LL   LL    LLLL     0 + 0 + 0 + 0 + 0 =  0
 LH   LH   LH   LH   LH    LHLH     1 + 1 + 1 + 1 + 1 =  5
 HL   HL   HL   HL   HL    HLHL     2 + 2 + 2 + 2 + 2 = 10
 HH   HH   HH   HH   HH    HHHH     3 + 3 + 3 + 3 + 3 = 15
-------------------------------------------------------------
```

4

DESCRIPTION

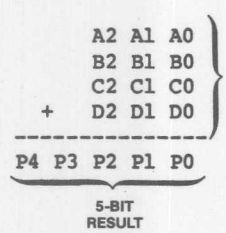
THIS PLE10P4 PERFORMS PARTIAL PRODUCTS REDUCTION FOR WALLACE TREE COMPRESSION. FIVE ROWS OF 2-BIT NUMBERS (A1-A0, B1-B0, C1-C0, D1-D0, AND E1-E0) ARE NUMERICALLY SUMMED TO PRODUCE A 4-BIT RESULT (P3-P0).

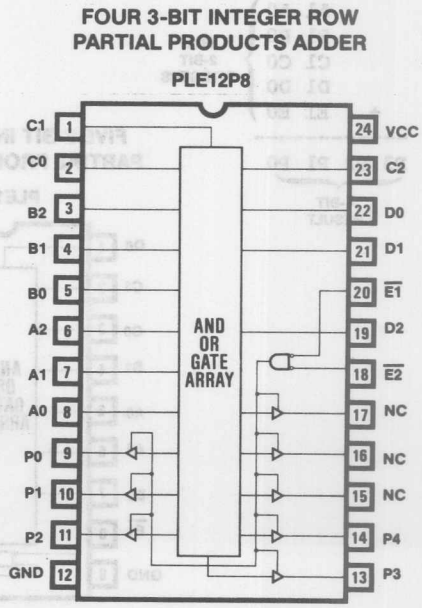


```
PLE12P8                                          PLE DESIGN SPECIFICATION
P5021                                              VINCENT COLI 02/10/83
FOUR 3-BIT INTEGER ROW PARTIAL PRODUCTS ADDER
MMI SANTA CLARA, CALIFORNIA
.ADD A0 A1 A2 B0 B1 B2 C0 C1 C2 D0 D1 D2
.DAT P0 P1 P2 P3 P4


P4,P3,P2,P1,P0 = A2,A1,A0 .+. B2,B1,B0 .+. C2,C1,C0 .+. D2,D1,D0  ; P = A+B+C+D


FUNCTION TABLE

A2 A1 A0 B2 B1 B0 C2 C1 C0 D2 D1 D0 P4 P3 P2 P1 P0

;AAA   BBB   CCC   DDD   PPPPP     COMMENTS
;210   210   210   210   43210     A + B + C + D =  P
-------------------------------------------------------
 LLL   LLL   LLL   LLL   LLLLL     0 + 0 + 0 + 0 =  0
 LLH   LLH   LLH   LLH   LLHLL     1 + 1 + 1 + 1 =  4
 LHL   LHL   LHL   LHL   LHLLL     2 + 2 + 2 + 2 =  8
 LHH   LHH   LHH   LHH   LHHLL     3 + 3 + 3 + 3 = 12
 HLL   HLL   HLL   HLL   HLLLL     4 + 4 + 4 + 4 = 16
 HHH   HHH   HHH   HHH   HHHLL     7 + 7 + 7 + 7 = 28
-------------------------------------------------------
```

DESCRIPTION

THIS PLE12P8 PERFORMS PARTIAL PRODUCTS REDUCTION FOR WALLACE TREE COMPRESSION. FOUR ROWS OF 3-BIT NUMBERS (A2-A0, B2-B0, C2-C0, AND D2-D0) ARE NUMERICALLY SUMMED TO PRODUCE A 5-BIT RESULT (P4-P0).

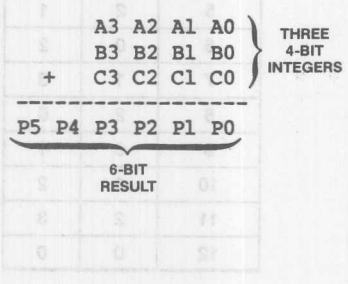




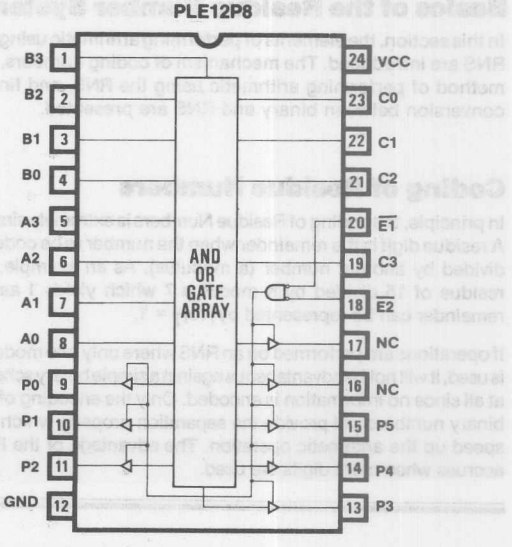
```
PLE12P8                                          PLE DESIGN SPECIFICATION
P5022                                              VINCENT COLI 08/10/83
THREE 4-BIT INTEGER ROW PARTIAL PRODUCTS ADDER
MMI SANTA CLARA, CALIFORNIA
.ADD A0 A1 A2 A3 B0 B1 B2 B3 C0 C1 C2 C3
.DAT P0 P1 P2 P3 P4 P5

P5,P4,P3,P2,P1,P0 = A3,A2,A1,A0 .+. B3,B2,B1,B0 .+. C3,C2,C1,C0  ; P = A+B+C

FUNCTION TABLE

A3 A2 A1 A0 B3 B2 B1 B0 C3 C2 C1 C0 P5 P4 P3 P2 P1 P0

;AAAA   BBBB   CCCC   PPPPPP      COMMENTS
;3210   3210   3210   543210      A  +  B  +  C  =  P
-------------------------------------------------------
 LLLL   LLLL   LLLL   LLLLLL     0  +  0  +  0  =  0
 LLLH   LLLH   LLLH   LLLLHH     1  +  1  +  1  =  3
 LLHL   LLHL   LLHL   LLLHHL     2  +  2  +  2  =  6
 LHLL   LHLL   LHLL   LLHHLL     4  +  4  +  4  = 12
 HLLL   HLLL   HLLL   LHHLLL     8  +  8  +  8  = 24
 HHHH   HHHH   HHHH   HLHHLH    15  + 15  + 15  = 45
-------------------------------------------------------
```

DESCRIPTION

THIS PLE12P8 PERFORMS PARTIAL PRODUCTS REDUCTION FOR WALLACE TREE COMPRESSION. THREE ROWS OF 4-BIT NUMBERS (A3-A0, B3-B0, AND C3-C0) ARE NUMERICALLY SUMMED TO PRODUCE A 6-BIT RESULT (P5-P0).



THREE 4-BIT INTEGER ROW PARTIAL PRODUCTS ADDER



Residue Arithmetic Using PLEs

Conventional binary arithmetic can be replaced by another kind of computational methodology known as the Residue Number System. The use of this system allows integer arithmetic to be performed by arrays of PLEs. The idea of PLEs as arithmetic elements is simply to store pre-computed values of the arithmetic operation in the PLE memory cells, and to use the input variables to the arithmetic as addresses to the PLE. Since we are pre-computing the results of the arithmetic operations, the same PLE may be used for many different functions. As an example, a 256 x 8-bit PLE can be used as a 4 x 4-bit binary multiplier, or a 4 + 4-bit binary adder with the output multiplied by any 3-bit constant. It is this flexibility which holds so much appeal for the use of PLEs as computational elements.

Introduction

Arithmetic operations often involve carry propagation. This propagation causes too much delay for high-speed arithmetic. The Residue Number System (RNS) provides the required separation property needed for high-speed arithmetic. Each digit of the RNS representation is coded into a certain number of bits. In performing the basic operations of addition, subtraction, and multiplication, no information is required to be passed between the digits. Therefore, the number of bits required for representing each digit can be partitioned so that commercially available PLEs can be used to implement the arithmetic.

Basics of the Residue Number System

In this section, the elements of performing arithmetic using the RNS are introduced. The mechanism of coding numbers, the method of performing arithmetic using the RNS, and finally conversion between binary and RNS are presented.

Coding of Residue Numbers

In principle, the coding of Residue Numbers is extremely simple. A residue digit is the remainder when the number to be coded is divided by another number (a modulus). As an example, the residue of 15 divided by a modulus 7 which yields 1 as the remainder can be represented by $|15|_7 = 1$.

If operations are performed on an RNS where only one modulus is used, it will not be advantageous against a simple binary scheme at all since no information is encoded. Only the encoding of the binary numbers will provide the separation property which will speed up the arithmetic operation. The advantage of the RNS accrues when more digits are used.

Another example of encoding a number using 3 moduli to give a 3-digit RNS representation is as follows: let the moduli be m1 = 3, m2 = 4, m3 = 5. The residues of X = 25 will be shown as xi where i = 1, 2, 3. Thus,

$$X1 = |25|_{m1} = |25|_3 = 1$$
$$X2 = |25|_{m2} = |25|_4 = 1$$
$$X3 = |25|_{m3} = |25|_5 = 0$$

In the RNS using the moduli 3, 4, 5, the number 25 is represented as (1, 1, 0).

The number of unique representations for a set of moduli is the Least Common Mulitple (LCM) of the moduli. The most efficient set of moduli is one in which all moduli are pairwise relatively prime.

Tables 1 illustrates an example of a set of moduli (3, 4) which can represent 12 integers. Note that the representations of 0 and 12 are the same, since the representation repeats itself after 12 integers.

X	(3) x1	(4) x2
0	0	0
1	1	1
2	2	2
3	0	3
4	1	0
5	2	1
6	0	2
7	1	3
8	2	0
9	0	1
10	1	2
11	2	3
12	0	0

Table 1. Representation of 0 to 12 in RNS using Moduli 3 and 4. The representation repeats itself after 12 integers

In table 2, (4, 6) is the set of moduli uses. Since 4 and 6 are not relatively prime, the number of integers that can be represented is not the product of 4 and 6, but instead is the LCM of 4 and 6 which is 12. The representation again repeats itself once every 12 integers.

X	(3) x1	(4) x2
0	0	0
1	1	1
2	2	2
3	3	3
4	0	4
5	1	5
6	2	0
7	3	1
8	0	2
9	1	3
10	2	4
11	3	5
12	0	0
13	1	1
14	2	2
15	3	3
16	0	4
17	1	5
18	2	0
19	3	1
20	0	2
21	1	3
22	2	4
23	3	5
24	0	0

Table 2. Representation of 0 x 24 for Moduli 4 and 6. Since 4 and 6 are not relatively prime, and their LCM is only 12, the representation again repeats itself every 12 integers

Negative numbers are formed in the same way negative numbers are formed in binary (two's complement) system. To form the two's complement of a number in binary, we subtract the number 2^B where B is the number of bits of the representation. In RNS, we subtract the RNS number from mi to form the negative. Table 1 can be rewritten as in table 3 for encoding of negative numbers.

X	(3) x1	(4) x2
0	0	0
1	1	1
2	2	2
3	0	3
4	1	0
5	2	1
-6	0	2
-5	1	3
-4	2	0
-3	0	1
-2	1	2
-1	2	3

Table 3. Representation of -6 to 5 in RNS using Moduli 3 and 4

Arithmetic Using the RNS

For two RNS numbers, X and Y, the result of the addition of the two numbers, Z, in RNS is given by:

$|xi + yi|_{mi} = zi$ for all of the RNS digits.

The same result is found for subtraction and multiplication. This means that arithmetic can be carried out between the same digits of the two numbers, X and Y, without interaction between adjacent digits. The arithmetic is therefore "carry-free". As an example, let us consider the following computation:

$Z = (863 \times 3942) + (-862 \times 3942) = 3942$

We only need sufficient dynamic range to represent the result; intermediate overflows can be ignored. Let us choose the following moduli for the RNS representation:

m1 = 7, m2 = 9, m3 = 11, m4 = 13

M = 9009

The above set can represent numbers in the range -4505 to 4504, and so this number range is sufficient for the calculation of this example. The computation is shown in table 4.

X	(7) x1	(9) x2	(11) x3	(13) x4	
3942	1	0	4	3	
863	2	8	5	5	
862	1	7	4	4	
-862	6	2	7	9	
863 x 3942	2	0	9	2	
-862 x 3942	6	0	6	1	
Z	1	0	4	3	= 3942

Table 4. Calculating Z = 863 x 3942 + (-862 x 3942) = 3942

Division and Scaling

Division of residue numbers is more complicated than addition, subtraction, or multiplication. If the dividend is exactly divisible by the divisor, the operation is easier. In this case, a division by a number is the same as a multiplication by the inverse of that number. The multiplication inverse of an integer X in modulo arithmetic can be found by finding the vector (d1, . . . ,dn) which satisfies the following:

$$|X.di|_{mi} = 1$$

For example, 95 divided by 5 in moduli 2, 7 and 9 can be done by first finding the vectors representing 95 and the inverse of 5.

$$|95|_2 = 1$$
$$|95|_7 = 4$$
$$|95|_9 = 5$$

So, for the multiplicative inverse of 5, we have:

$$|1/5|_2 = 1 \qquad |5 \times 1|_2 = 1$$
$$|1/5|_7 = 3 \text{ since } |5 \times 3|_7 = 1$$
$$|1/5|_9 = 2 \qquad |5 \times 2|_9 = 1$$

Therefore, $|95/5|_2 = ||95|_2 \times |1/5|_2|_2 = |1 \times 1|_2 = 1$

$$|95/5|_7 = ||95|_7 \times |1/5|_7|_7 = |4 \times 3|_7 = 5$$
$$|95/5|_9 = ||95|_9 \times |1/5|_9|_9 = |5 \times 2|_9 = 1$$

and the answer is 19.

The operation becomes more complicated when the dividend is not exactly divisible by the divisor or one of the moduli of the multiplicative inverse does not exist, say, if the residue of the divisor for that modulus is 0. In this case, we need to obtain the remainder and then subtract the remainder from the dividend and then perform the division. The problem in finding the remainder seems to be the same as performing the division itself. However, this type of division can be done in a process called scaling, which will not be discussed in detail in this paper.

In spite of the improvements made in implementing scaling algorithms, scaling still represents a major effort in any calculation. It is advisable to use RNS only on systems where many arithmetic operations can be performed for each scaling operation.

A System Using an RNS

An RNS is very useful in systems which has predefined operations and dynamic ranges. Moreover, it can only operate on integers, or at most, block floating-point numbers. Since the RNS involves conversions between integers and their RNS representations, and conversions by themselves are already time-consuming, the problem to be solved in the RNS system should be operation intensive.

Figure 1. Architecture of an RNS

Conversion to RNS Representation

The conversion of an integer to RNS can be viewed as a mapping process. PLEs are naturally implementation for mapping. For example, if an 8-bit integer is used to represent numbers ranging from 0 to 255, and the following moduli are arbitrarily chosen for conversion to RNS — 2, 11 and 15 (which can represent 330 integers), 8 bits of address are needed for the integer input and 9 outputs (1 for modulus 2, 4 for modulus 11 and 4 for modulus 15). In reality, only 8 outputs are needed because that bit of residue for modulus 2 is not required, since the least significant bit of the integer is also the residue of itself in modulus 2. In fact, a PLE8P8 will be sufficient.

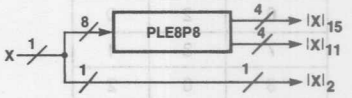


Figure 2. Mapping an 8-bit integer X to its residues on moduli 2, 11, and 15

Another example is a 14-bit integer which is to be converted to RNS. A 14-bit address needs 16 K address spaces for the mapping. 16 K is too deep for a PLE. An alternative is to use 4 K-deep PLEs: PLE12P4s and PLE12P8s and a selector (e.g. a PLE5P8 to control the PLEs (see Figure 3). The PLE5P8 will decode two of the address bits and will selectively enable one of the four sets of PLEs as the mapping set, thus deepening the effective address to 16 K.

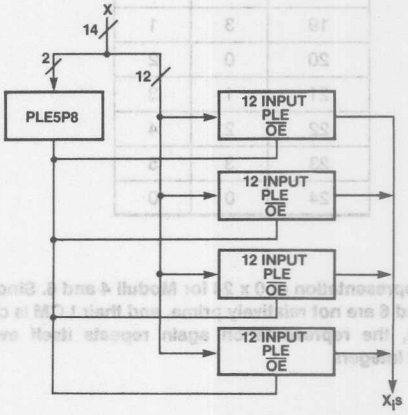


Figure 3. Mapping a 14-bit integer X to its residues by selectively enabling the outputs of one of the four sets of 12-input PLEs

This method of expansion is not effective with bigger integers. If the integer is N-bit and the address space of the PLEs available is M-bit, then 2^{N-M} sets of PLEs will be needed. Besides, as the dynamic range increases, the width of the outputs will also increase about proportionally. An alternative method is to use 2 or more levels of PLEs to generate the residues. The first level generates the remainders from the more significant bits of the integer and the products of some of the moduli. These remainders are in turns concatenated with the rest of the bits to become the inputs to the second level PLEs.

For example, for a 16-bit integer 43689, and let us use (2, 11, 13, 15, 23) as the set of moduli. We may choose 23, 30 and 143 as the moduli for the first level. The first level consists of PLE12P4s and PLE12P8s which generate the remainders of the most significant 12 bits of 43689 which is 2730. We know that $|2730|_{23}$ will be at most 22 and can therefore be represented by a 5-bit number; $|2730|_{15}$ will be at most 14 and can be represented by another 4-bit number; and $|2730|_{143}$ will be at most 142 and can be represented by a 6-bit number. The 5-bit number represented by $|2730|_{23}$ will be concatenated with the least significant 4 bits of the integer and gives a 9-bit number which can perform another division by 23 to give the final $|43689|_{23}$; the 4-bit number represented by $|2730|_{15}$ will be concatenated with the least significant 4 bits of the integer and gives an 8-bit number which can perform another division by 15 to give the final $|43689|_{15}$; the 6-bit number represented by $|2730|_{143}$ will be concatenated with the least siginificant 4 bits of the integer and gives a 10-bit number which can perform another division by 11 and 13 to give the final $|43689|_{11}$ and $|43689|_{13}$. As in the first example, $|43689|_2$ is just the least significant bit of the integer.

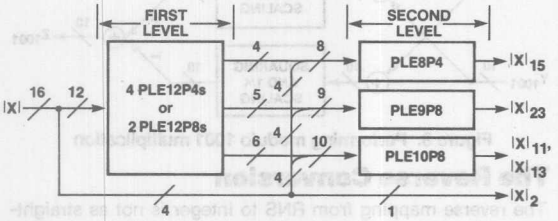


Figure 4. Mappings a 16 bit Integer X to residues in modulo 2, 11, 13, 15, and 23 using two-level mapping. The first level gives remainders from the more significant 12 bits, while the second level finds the final residues

In some circumstances, although an N-bit integer only has a dynamic range of 2^N, the intermediate calculations may overflow. It is sometimes necessary to add some other moduli to boost up the dynamic range for the intermediate calculations.

Arithmetic Operations In RNS

The arithmetic operations of the RNS is different from regular arithmetic in that even simple addition must be performed in modulo arithmetic. Simple ALU may not be able to handle this arithmetic. Again, PLEs are proven to be most useful. A PLE8P4 can perform addition, subtraction, or multiplication on two 4-bit residue numbers and give a 4-bit modulo result.

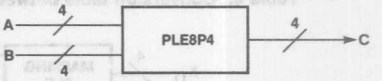


Figure 5. Calculating C = A + B, A - B, B - A, or A x B using a PLE

If the modulus is large, say greater than 64, the combined number of bits for two residues will be greater than the number of address bits for the largest of the commercially available PLEs. Of course, more than one PLEs can be used to depthen the effective address space. In this case, for every additional bit of a modulus, two more bits of address will be needed — one for each operand. In other words, for each additional bit of a modulus, the address space of operation will be quadrupled. It is not very effective when the modulus grows too large. Fortunately, for both addition and multiplication, there are more efficient procedures.

Large Modulus Addition

Table 5 shows the contents required for the addition operations in modulus 11. There is a lot of redundancy in the table which can be compressed by reducing what should be 8-bits of inputs to 5-bits. What we need is just another level of mapping. There are a total of 121 combinations for a number of modulus 11 operate on another operand of the same modulus. In real, only numbers ranging from 0 to 10 can be represented in modulus 11. The sum ranges from 0 to 20 (not in modulus 11). This range can be represented by a new set of submoduli (3, 7) which is 5-bits wide. In fact, any new set of submoduli which has a dynamic range of at least 21 can be used. The operands in modulus 11 will be converted to their representations in submoduli 3 and 7. The addition is done in the submoduli and the result is reconverted back to modulus 11 RNS (see Table 6).

x_{11} 4

y_{11} 4

	0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10	0
2	2	3	4	5	6	7	8	9	10	0	1
3	3	4	5	6	7	8	9	10	0	1	2
4	4	5	6	7	8	9	10	0	1	2	3
5	5	6	7	8	9	10	0	1	2	3	4
6	6	7	8	9	10	0	1	2	3	4	5
7	7	8	9	10	0	1	2	3	4	5	6
8	8	9	10	0	1	2	3	4	5	6	7
9	9	10	0	1	2	3	4	5	6	7	8
10	10	0	1	2	3	4	5	6	7	8	9

Table 5. Addition table in modulo 11 arithmetic

|x + y| 11	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7	8	9
|x + y| 7	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5	6
|x + y| 3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2

Table 6. Conversion table between modulo 11 arithmetic and modulo 3 and 7 arithmetic

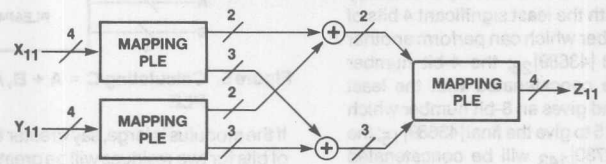


Figure 6. Calculating addition of two numbers in modulo 11 using submoduli operations

Large Modulus Multiplication

The solution to this problem in multiplication is similar. For example, if two RNS digits in modulus 91 is to be multiplied, (7, 13) may be chosen as a set of submoduli. The representation of an RNS digit in modulus 91 needs 7-bits. These 7-bits are first mapped to two RNS digits — in modulo 7 which needs 3-bits; and in modulo 13 which needs 4-bits. The representations of the two operands in the two moduli can then be multiplied and give the result in modulo 7 and modulo 13. The result is then converted back to modulo 91. Unfortunately, this scheme can be used only when the modulus can be expressed as a product to two integers which are relatively prime. But, in this case, the RNS digit may simply be represented as the residue of the two smaller integers instead of using them as submoduli.

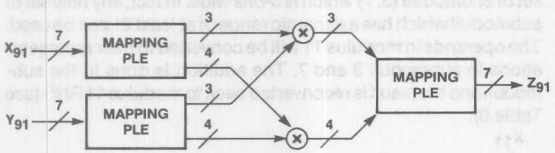


Figure 7. Calculating multiplication of two numbers in modulo 91 using submoduli operations

Suppose another modulus 101 is used. 101 is a prime number and RNS in modulus 101 ranges from 0 to 100. The real dynamic range of the product of two numbers in modulus 101 is 0 to 10000, which is already too large for an address space for a PLE. For this modulus, we may use three 4 K-deep PLEs to deepen the address space. For a modulus like 1001, it may not be too efficient to use this scheme. Instead, since:

$$xy = [(x + y)^2 - (x - y)^2] / 4$$
$$\text{or} \quad = [x + y)^2] / 4 - [(x - y)^2] / 4$$

we may do x + y and x − y first and then do the squaring of the sum and the difference scaled by a factor of 4. Since the final product of two integers must be an integer, the squaring and scaling may be performed in one operation with the fractional part discarded. The way to obtain x + y and x − y is the same as what was discussed earlier in the "Large Modulo Addition" session.

In any event, operations on residues of large moduli are slower and involve more hardware and are not recommended.

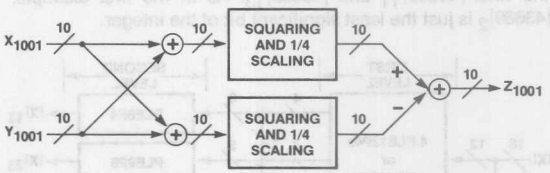


Figure 8. Performing modulo 1001 multiplication

The Reverse Conversion

The reverse mapping from RNS to integer is not as straightforward as the other way. For an RNS system which has a total of 12 bits for all the residues, we can still use 12-input PLEs to convert. We may also use several sets of 12-input PLEs to reverse map the RNS if the integer is not much longer. But for very long integers, we may need to use the general algorithm for the reverse map:

1. Find $M = m1 \times m2 \times \ldots \times m_{n-1}$ (where n is the number of moduli)
2. Find $ti = M/mi$
3. Find $X = |x1t1 + x2t2 + \ldots + xn{-}1tn{-}1|M$

In hardware implementation, ti's are all known beforehand. We can map xi's to get the xiti's. Then we may perform Wallace Tree Compression (see the session on this subject in this handbook for more information) on the xiti's to give two-level operands which add to the final sum and divide it by M to get X. Again, PLEs provide the best solution for Wallace Tree Compression.

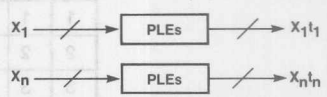


Figure 9a. Reverse mapping to get X_it_i.

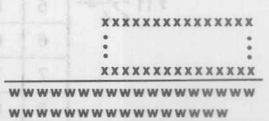


Figure 9b. Modulo M Wallace tree compression to reduce the number of levels for summation to 2 followed by an addition and division to get $X = |x_1t_1 + \ldots + x_nt_n| M$

Conclusion

Memory elements provide excellent solution to mapping functions — for control purposes, for arithmetic operations and general logic replacements. This paper investigates the possibility of using PLEs as arithmetic units. In fact, for logic like residue number arithmetic, there is no better solution than to use PLEs.

Acknowledgement

Portions of this article were extracted from *"Integer Arithmetic Using PROMs"* by Dr. G. A. Jullien of the University of Windsor, Canada.

Distributed Arithmetic Using PLEs

Distributed Arithmetic using PLEs

In digital signal processing, sum-of-product type of operations are often necessary. These operations take the form of:

$$y = \sum_{i=1}^{M} a_i x_i \qquad \text{where } a_i\text{'s are some constants}$$

If real multiplications are to be performed on every product term, it will need a total of M multiplications and M-1 additions. Multiplication operations normally take much longer than simple addition. An alternative to calculate equations of the above form is by using distributed arithmetic.

Suppose there is an N-bit integer X given by :

$$x = [x(N-1), x(N-2), \ldots, x(1), x(0)]$$

or equivalently:

$$x = \sum_{j=0}^{N-1} x(j)\, 2^j$$

where x(N-1) is the most significant bit. The equation:

$$y = \sum_{i=0}^{M} a_i x_i$$

can be expressed as:

$$y = \sum_{i=1}^{M} a_i \left(\sum_{j=0}^{N-1} x_i(j)\, 2^j \right)$$

$$= \sum_{j=0}^{N-1} 2^j \left(\sum_{i=1}^{M} a_i x_i(j) \right)$$

Now, let:

$$H(j) = \sum_{i=1}^{M} a_i x_i(j)$$

Since H (j) is independent of i and since a_1's are all constants, we precompute for every x (j) = [x_1 (j), x_2 (j), . . . , x_M (j)] the values of H (j). Then x (j) can be used as the address of PLEs whose outputs are the precomputed result H (j).

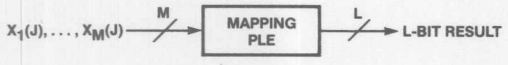


Figure 1. Mapping the j^{th} bit from each of the x_i's to an L-bit result

If there are M bits of data and the result is L-bit wide, and if M is very large, say 20, and L is 8, then we need 20 bits of address lines if we want to use only PLE mapping. Since 20 bits of address translate to 1M words, and there is no available 1M-deep PLE on the market, it is not realistic to use PLE mapping. Instead, H (j) can be partitioned as follows:

$$H(j) = \sum_{i=1}^{20} a_i x_i(j) \qquad \text{for } M = 20$$

$$= \sum_{i=1}^{10} a_i x_i(j) + \sum_{i=11}^{20} a_i x_i(j)$$

the 20-bit address can be separated to two 10-bit addresses and each of them is individually mapped. The two outputs will then be added together to give H (J). An implementation of this mapping is shown in figure 2.

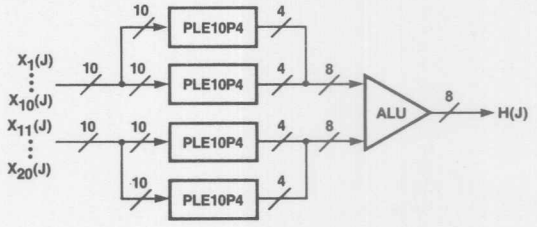


Figure 2. Mapping the j^{th} bit of each of x_i's to an L-bit result when there are too many x's (20 in this case)

There is another alternative for implementing a sum-of-product operation: by using a multiplier accumulator (MAC).

The main constraint on distributed arithmetic is that one set of the multiplicands must be fixed, i.e. ai's in this case, for the sum-of-product mapping while a MAC will allow flexibility.

There are normally some constraints on the width for the data bus from which the operands are loaded. If all the operands are new, it will need M cycles to load in the operands anyway, distributed arithmetic offers no advantages over MAC since distributed arithmetic needs to wait for all the operands to be loaded in before any operation can start while MAC can perform a multiplication and an addition every cycle. M cycles will be needed anyway for the complete operation using a MAC while distributed arithmetic may take even longer.

On the other hand, for operations like convolutions where one set of operands are fixed and only one new variable operand is needed for every result, distributed arithmetic will be a better solution since it can give a result in every clock-cycle while a MAC will need M-cycles (because recalculations of all the product terms are necessary). An implementation for convolution is shown in Figure 3.

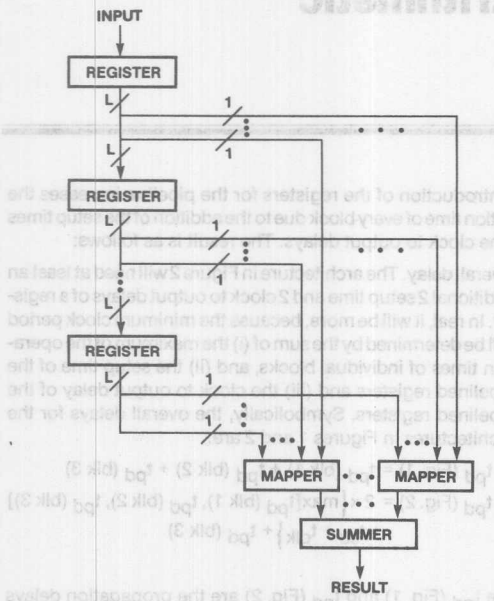


Figure 3. An implementation of a distributed arithmetic system for convolution

There is another way to implement distributed arithmetic through bit-serialization:

From H (j), the sum-of-product of y can be obtained as:

$$y = \sum_{j=0}^{N-1} 2^j H(j)$$

To implement this equation, consider that the least significant bit of the result is to be used only for rounding purposes only. Only the more significant bits will be retained. The computation can be performed in the following way:

1) For j = 0,
   $y_0 = 2^0 H(0) = H(0)$
2) For j = 1 to N–1
   $y_j = H(j) + 1/2\ H(j-1)$

Note that the second term of the last equation means that the previous result (y_{j-1}) is shifted right one-bit; the last bit of y_{j-1} is truncated.

The implementation of such a system is shown in Figure 3. The system consists of a shift register, a mapper (PLEs, or PLEs with adders), an accumulator, and an ALU.

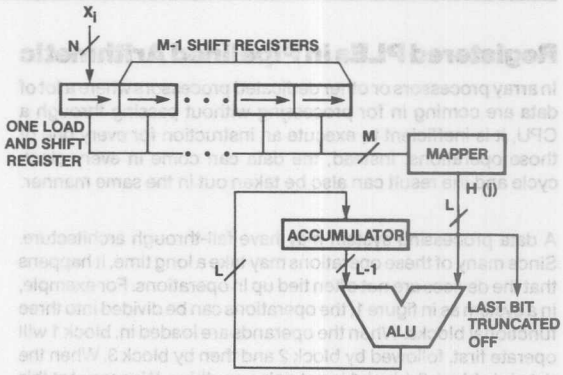


Figure 4. A bit-serializations implementation for a distributed arithmetic system

The operations are as follows:

1) Load x_j onto the load and shift register at clock 0.
2) Load H (0) onto accumulator and shift all registers at clock 1.
3) From clock k (between clock 2 to clock N–1), the content of the accumulator will be replaced by the sum of H (k–1) and the more significant N–1 bits of the current accumulator value.
4) For clock N, the following are performed:
   a) Repeat step 3. At the end of the operation, the accumulator contains the value of the result (scaled by the number of shifts).
   b) x_{j+1} is loaded onto the load and shift register. The shifting frequency is equal to N times the basic rate.

Due to the fact that there are a number of shift operations necessary for each data load, this method is recommended for the following conditions:

1) This design is under cost, power dissipation, and board space constraints.
2) This design is for high M-to-N ratio-array multiplications.

Registered PLEs in Pipelined Arithmetic

In array processors or other dedicated processors where a lot of data are coming in for processing without passing through a CPU, it is inefficient to execute an instruction for every one of those operations. Instead, the data can come in every clock cycle and the result can also be taken out in the same manner.

A data processing system may have fall-through architecture. Since many of these operations may take a long time, it happens that the devices are not often tied up in operations. For example, in a system as in figure 1, the operations can be divided into three functional blocks. When the operands are loaded in, block 1 will operate first, followed by block 2 and then by block 3. When the data is in block 2, block 1 is not doing anything. We cannot at this time put in the next set of operands because changes in operands may disturb the operation in block 2.

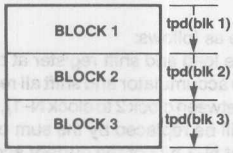


Figure 1. An example of the fall-through approach to arithmetic operation

A solution to this is by registering the operands and signal paths when the operations is switched to block 2; and by registering the operands and signal paths again when the operations is carried out in block 3. The result is stated in figure 2. This architecture is called the pipelined structure. It makes the loading of the second set of operands possible even before the first result is out, thus increasing the throughput.

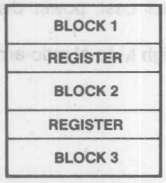


Figure 2. Pipelined arithmetic operation

The introduction of the registers for the pipeline increases the operation time of every block due to the addition of the setup times and the clock to output delays. The result is as follows:

1) Overall delay. The architecture in Figure 2 will need at least an additional 2 setup time and 2 clock to output delays of a register. In real, it will be more, because the minimum clock period will be determined by the sum of (i) the maximum of the operation times of individual blocks, and (ii) the setup time of the pipelined registers and (iii) the clock to output delay of the pipelined registers. Symbolically, the overall delays for the architectures in Figures 1 and 2 are:

$$t_{pd}\ (\text{Fig. 1}) = t_{pd}\ (\text{blk 1}) + t_{pd}\ (\text{blk 2}) + t_{pd}\ (\text{blk 3})$$

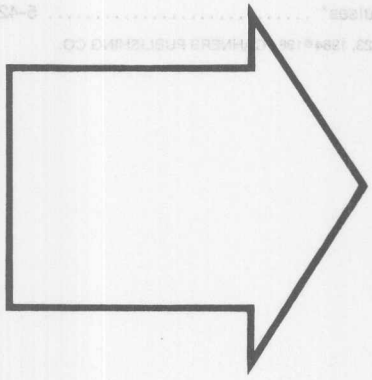
$$t_{pd}\ (\text{Fig. 2}) = 2 \times \left\{ \max[t_{pd}\ (\text{blk 1}), t_{pd}\ (\text{blk 2}), t_{pd}\ (\text{blk 3})] + t_{su} + t_{clk} \right\} + t_{pd}\ (\text{blk 3})$$

Where t_{pd} (Fig. 1) and t_{pd} (Fig. 2) are the propagation delays of the architectures in figure 1 and figure 2 respectively; t_{pd} (blk 1) t_{pd} (blk 2), t_{pd} (blk 3) are the propagation delays of block 1, block 2, and block 3 respectively; and t_{su} and t_{clk} are the setup time and clock-to-output delay of the registers respectively.

2) Throughputs of clock rate. The architecture in figure 1 has a throughput period of (t_{pd} (blk 1) + t_{pd} (blk 2) + t_{pd} (blk 3) + t_{su} + t_{clk}), assuming that the operands are coming from and the result is going to some registers; the architecture in Figure 2 has a throughput period of (max[t_{pd} (blk 1), t_{pd} (blk 2), t_{pd} (blk 3)] + t_{su} + t_{clk}) which is faster.

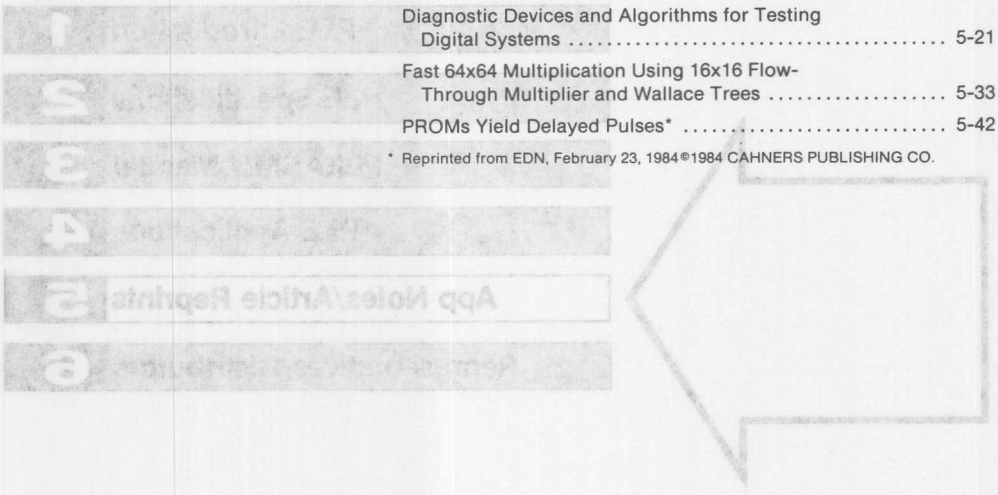
PLEs are useful as logic elements, and registered PLEs are excellent media for pipelined arithmetic. Monolithic Memories supplies a number of registered PLEs which provide effective solutions to pipelined systems.

Applications for pipeline arithmetic include array and digital signal processing.



Contents of Section 5

* Reprinted from EDN, February 23, 1984 ©1984 CAHNERS PUBLISHING CO.



High-Speed Bipolar PROMs Find New Applications As Programmable Logic Elements

Vincent J. Coli and Frank Lee
9th West Coast Computer Faire

Classic applications for bipolar PROMs include instruction storage for microprogram control store and software for microprocessor programs. However, due to a new design methodology and state-of-the-art performance, PROMs are finding increasing numbers of applications as Programmable Logic Elements (or PLEs™). This paper will cover the architecture, applications, and software support for PLEs.

Fuse-Programmable Logic Families

A typical combinatorial Boolean equation can be written in sum-of-product form, which consists of several AND gates summed at an OR gate. In general, a set of combinatorial Boolean equations with n inputs (I0, I1, . . . , In–1) and m outputs (O0, O1, . . . , Om–1) can be generated through one level of AND gates followed by one level of OR gates. Custom logic functions can be defined using programmable logic.

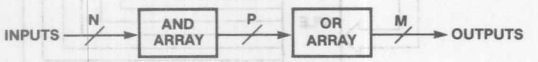


Figure 1. Structure of Programmable Logic Devices

Fuse-programmable devices normally consist of two levels of logic — AND-array and OR-array — as suggested above. There are three basic types of fuse-programmable devices — PROM (Programmable Read Only Memory), PLA (Programmable Logic Array), and PAL (Programmable Array Logic). Which arrays are fuse-programmable distinguish these three types of devices.

PLAs offer the greatest flexibility since both the AND and OR arrays are programmable. This flexibility comes with the cost of lower performance, higher power dissipation, and generally higher price.

A PAL device has only the AND-array programmable; the OR-array is fixed. Each output has an OR gate associated with it which sums a fixed number of product terms (AND combinations). Statistically there is only a limited number of product terms in any equation. So the flexibility of a PLA is normally not needed. This is a compromise between flexibility and cost and performance.

The OR-array is programmable in a PROM, but the fixed AND-array consists of all combinations of literals for each of the input variables. For example, there are 32 product terms available in a PROM with 5 inputs a,b,c,d,e (corresponding to words 0 through 31 in the PROM memory):

```
/a*/b*/c*/d*/e     (Word  0)
/a*/b*/c*/d* e     (Word  1)
/a*/b*/c* d*/e     (Word  2)
      .                .
      .                .
      .                .
a* b* c*/d* e      (Word 29)
a* b* c* d*/e      (Word 30)
a* b* c* d* e      (Word 31)
```

where '*' represents the Boolean AND operator and '/' represents the Boolean NOT or inverter operator. The fuses in the OR-array are programmed to select the desired AND combinations.

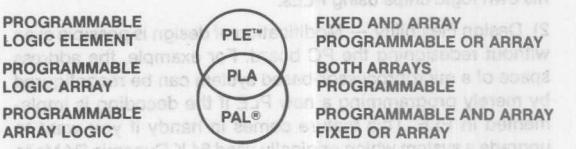


Figure 2. Structural differences between PLE (PROM), PLA and PAL. Note that the PALs and PLEs complement each other. The PAL has many input terms while the PLE is rich in product terms

The existence of all combinations of literals for all inputs makes it possible to define functions which cannot be implemented in a PLA or a PAL. For example, a 5-input Exclusive-OR (XOR) function can be implemented using 16 product terms. This may exceed the number of product terms available in a PAL and will consume too many product terms in a PLA, but can be constructed quite efficiently in a PROM. It is important to realize that any combination of inputs can be decoded in a PROM as long as sufficient input pins are provided since a PROM provides 2^n product terms (where n is the number of inputs). Another way of looking at this is that PROMs store the logic transfer function in a memory. The fixed AND-array (or AND-plane) consists of the row and column decoders while the fuses in the OR-array (or OR-plane) are the bits in the memory. In a memory, a fuse blown versus a fuse intact distinguishes a HIGH from a LOW.

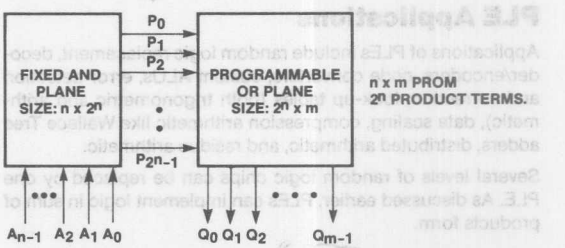


Figure 3. Block diagram of a PROM viewed as a PLE. Notice that the PLE provides many (2^n, where n is the number of inputs) product terms. A by-product of this is programmable output polarity: either Active-High or Active-Low output polarities are available

5

PROMs are also often used as logic devices. In this paper, PROMs are referred to as PLEs (Programmable Logic Elements).

Advantages of PLEs

PLEs provide a cost-effective solution for many applications. Here are just some of the advantages of PLEs:

1) Customizable Logic — The designer is limited to standard functions if SSI/MSI devices are used. The designer can create his own logic chips using PLEs.

2) Design Flexibility — Modification of design is possible even without redesigning the PC board. For example, the address space of a microprocessor-based system can be reconfigured by merely programming a new PLE if the decoding is implemented in PLE. This feature comes in handy if you want to upgrade a system which originally used 64 K Dynamic RAMs to now use state-of-the-art 256 K Dynamic RAMs.

3) Reduce Errors — Errors are sometimes unavoidable and oftentimes quite expensive. Programmable devices make it easier and less expensive to correct errors.

4) Reduction of Printed Circuit Board Space —PLEs save PC board space since several SSI/MSI functions can be integrated into a single package.

5) Fast Turnaround Time — With existing commercial programmers and development software support, a prototype of the custom tailored PLE will be ready in just a few minutes.

PLE Applications

Applications of PLEs include random logic replacement, decoder/encoders, code converters, custom ALUs, error detection and correction, look-up tables (both trigonometric and arithmetic), data scaling, compression arithmetic like Wallace Tree adders, distributed arithmetic, and residue arithmetic.

Several levels of random logic chips can be replaced by one PLE. As discussed earlier, PLEs can implement logic in sum of products form.

many of these functions are application dependent. A standard 3-to-8 decoder/demultiplexer (74S138) can be used in decoding applications. But the decoding scheme may require several 3-to-8 decoder/ demultiplexers and additional SSI OR-gates. On the other hand, a PLE can be customized to perform the required decoding function with no additional gates. Simple decoders, such as those used for decoding memory chip selects from address lines, can be implemented in a PLE with 5 to 10 inputs. More complex decoding may require 8 to 12 inputs.

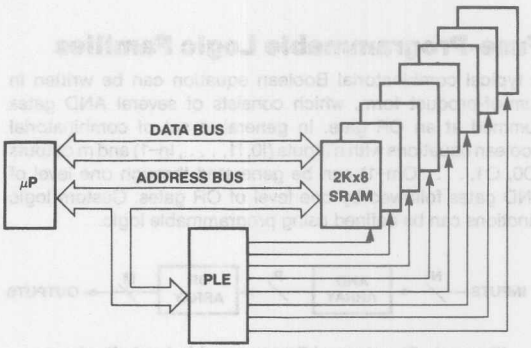


Figure 4. PLE address decoding application. The PLE selects one of eight 2Kx8 Static RAMs by decoding several microprocessor address lines

PLEs offer a very flexible solution for code conversion applications. Translations of codes such as from ASCII to EBCIDIC, Binary to BCD (Binary Coded Decimal), or BCD to Gray code can be implemented in PLEs. The 74S184 Binary-to-BCD Converter is actually a 32x8 PROM.

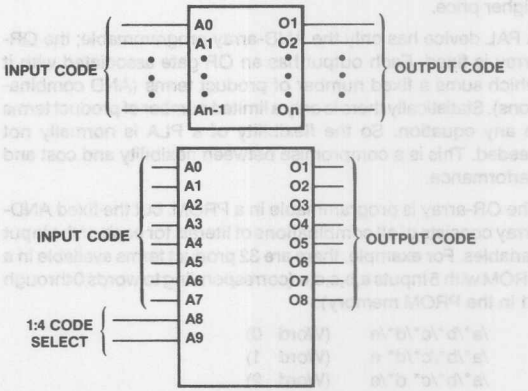


Figure 5. Two examples of PLE code converters. The second example illustrates how to use two inputs as code select lines so that four converters can be provided in one PLE

Standard ALUs (such as the 74S181) may not provide a very specialized function which a particular system requires, such as BCD arithmetic. In this case a PLE is again a good alternative. Although the PLE may be slower than a dedicated ALU, the presence of this specialized function is critical. For example, a 4-bit ALU can be constructed in a PLE with 12 inputs (A3-A0, B3-B0, I2-I0, Cin) and 8 outputs (F3-F0, /G, /P, Cout, A = B). Any 8 functions can be implemented.

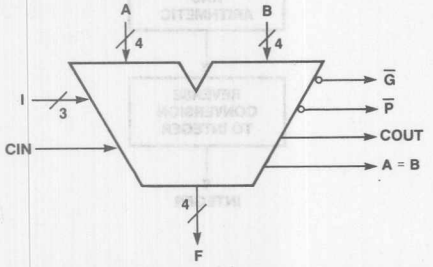


Figure 6. Block diagram for a 4-bit ALU which can be implemented in PLE

Data scaling is another PLE application. A dedicated multiplier is not required if the scaling factor is a constant; the prescaled result can be stored in a PLE. Fixed-bit multipliers are typically implemented in PLEs.

Column compression technique (also called Wallace Tree Compression) is used when expanding an array of several smaller parallel multipliers to perform large wordlength multiplication. These smaller multipliers will generate partial products (intermediate results) which must be numerically summed according to bit significance in order to calculate the final wordlength multiplication. Many levels of 2-input bus adders can be used to add these partial products, but the carry propagation delays may be too long. However, partial product adders implemented in PLEs can do compression of many levels without passing carries. Thus, the summation will be much faster.

"... THE 'S556, TOGETHER WITH PLEs ORGANIZED IN A WALLACE-TREE CONFIGURATION, CAN SAIL RIGHT ALONG AT THE RATE OF FOUR 56 X 56 MULTIPLICATIONS EVERY MICROSECOND ..."

Group Code Recorder (GCR) is an encoding/decoding scheme used for error detection on tape. During a WRITE operation, each 8-bit word is divided into two 4-bit nibbles. Both nibbles are then encoded into 5-bit codes before being recorded onto tape. Both 5-bit codes are decoded back to the original 4-bit nibbles and then combined during a READ operation. PLEs are exceptionally useful in mapping the 4-bit data to the 5-bit code and back.

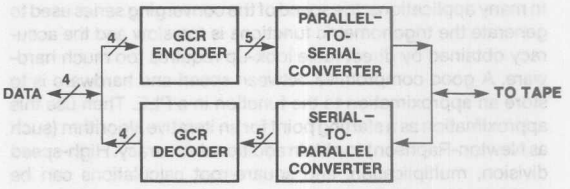


Figure 7. GCR encoder/decoder block diagram

Exclusive-OR gates, being half adders, are very prevalent in Error Detection and Correction (EDC) schemes. Many SSI chips are required to implement this function while PLAs and PALs may not provide sufficient product terms. PLEs are again an ideal solution.

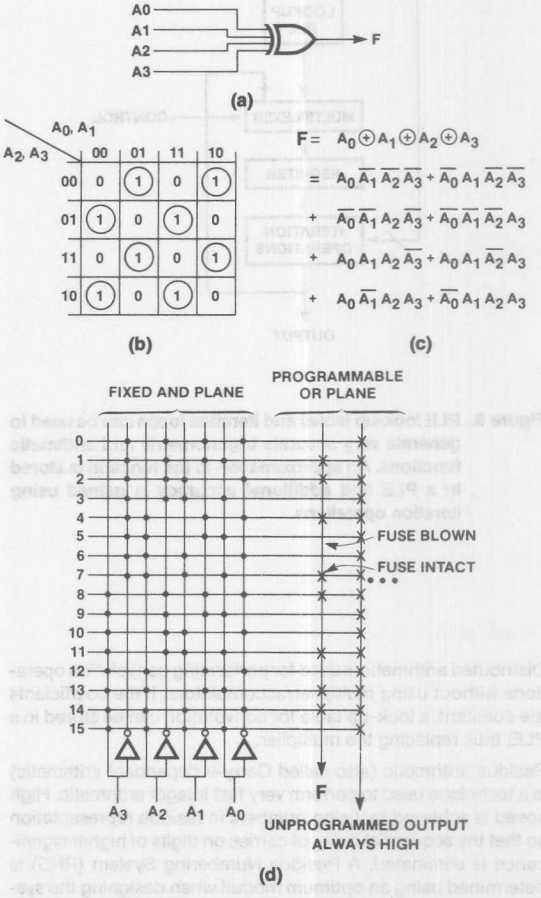


Figure 8. Exclusive-OR gates can be implemented in PLEs very efficiently. A 4-input XOR gate (a) maps into a checkerboard pattern in a Karnaugh Map (b) and requires eight product terms (c). The PLE implementation is shown in (d). An 8-input XOR gate requires sixteen product terms

5

In many applications, the speed of the converging series used to generate the trigonometric functions is too slow and the accuracy obtained by direct table look-up requires too much hardware. A good compromise between speed and hardware is to store an approximation to the function in a PLE. Then use this approximation as a starting point for an iterative algorithm (such as Newton-Raphson) to obtain additional accuracy. High-speed division, multiplication, and square-root calculations can be performed in a similar manner.

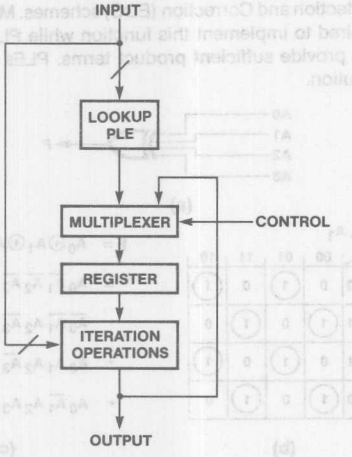


Figure 9. PLE look-up tables and iteration loops can be used to generate very accurate trigonometric and arithmetic functions. An approximation to the function is stored in a PLE and additional accuracy is gained using iteration operations

Distributed arithmetic is used for performing convolution operations without using multiplier/accumulators. If the coefficients are constant, a look-up table for convolution can be stored in a PLE, thus replacing the multiplier.

Residue arithmetic (also called Carry-Independent arithmetic) is a technique used to perform very fast integer arithmetic. High speed is achieved by using numbers in residue representation so that the sequential delay of carries on digits of higher significance is eliminated. A Residue Numbering System (RNS) is determined using an optimum moduli when designing the system. Conversion to and from residue representation are basic mapping functions which can be conveniently done in PLE. Also, since operations in residue arithmetic are performed using modulo addition and multiplication without carries, these operations can also be done using PLEs. In general, residue arithmetic should only be used for integer arithmetic which requires intensive operations.

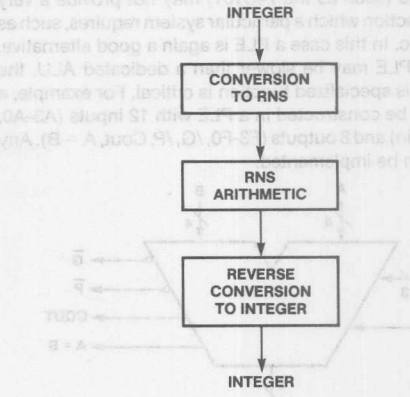


Figure 10. Architecture of a system based based on RNS. An integer number is converted to RNS representation using PLEs, then the RNS arithmetic is performed using PLEs, and finally the RNS result is converted back to integer representation again using PLEs

Restrictions

The basic restrictions for using PLEs to replace SSI/MSI parts are:

1) Since a memory element has a product term for every combination of literals of all the input terms, static hazard is normally unavoidable. For example, there are 5 inputs available in a 32 x 8 PROM. In order to generate a function like:

$$f = a^* b^* c^* d$$

The actual implementation inside the PROM will be:

$$f = a^* b^* c^* d^*/e + a^* b^* c^* d^* e$$

If a = b = c = d = HIGH, according to the first equation, we shall expect f to remain HIGH independent of e changing. In the actual PROM implementaton, there will be no hazard if e stays either HIGH or LOW. But if e changes, depending on whether e or /e will occur first, there exists the possibility that both product terms in the second equation will be LOW momentarily, which may cause a static logic hazard (HIGH to LOW to HIGH) for f. This hazard is commonly called a "glitch". Static hazards are not a problem for many applications, like those offered in this paper, but extreme care must be taken if the output of a PLE is used to strobe another device.

ADDRESS	e	d	c	b	a	f
00	0	0	0	0	0	0
01	0	0	0	0	1	0
02	0	0	0	1	0	0
.						
.						
.						
0C	0	1	1	0	0	0
0D	0	1	1	0	1	0
0E	0	1	1	1	0	0
0F	0	1	1	1	1	1
10	1	0	0	0	0	0
11	1	0	0	0	1	0
12	1	0	0	1	0	0
.						
.						
.						
1D	1	1	1	0	1	0
1E	1	1	1	1	0	0
1F	1	1	1	1	1	1

Figure 11. This Truth Table graphically illustrates the possible glitch (HIGH to LOW to HIGH hazard) for the function f = a* b* c* d implemented in a 32x8 PROM. Address 0F and 1F contain a 1 while all other locations contain a 0 for output f. If address input e should change, the PROM decoders could momentarily select a location containing a 0

2) Although PROMs (or PLEs) are available with registered outputs, internal feedback from the outputs and buried registers are not yet available in PROMs. External connections from some outputs to inputs must be made for applications which require feedback (such as in state machines). However Registered PROMs without feedback are useful for pipelining (overlap instruction fetch and execution) in order to increase system throughput.

PLEASM™ Software Support

Monolithic Memories has developed a software tool to assist in designing and programming PROMs as PLEs. This package, called "PLEASM" (PLE Assembler), is available for several computers including the VAX/VMS and IBM PC/DOS. PLEASM converts design equations (Boolean and arithmetic) into truth tables and formats compatible with PROM programmers. A simulator is also provided to test a design using a Function Table before actually programming the PLE. The PLEASM operators are listed below and the PLEASM catalog of operations is given on the next page. A sample PLE Design Specification (source code for PLEASM) with PLEASM outputs is given in Figure 12. PLEASM may be requested through the Monolithic Memories IdeaLogic Exchange.

Operators (in hierarchy of evaluation)

;	Comment follows
.	Dot operator (pin list or arithmetic operator follows)
ADD	Address pins (Inputs)
DAT	Data pins (Outputs)
,	Delimiter, separates binary bits (MSB first)
=	Equality (combinatorial)

BOOLEAN OPERATORS

/	Complement, prefix to a pin name
*	AND (product)
+	OR (sum)
:+:	XOR (exclusive or)
:*:	XNOR (exclusive nor)

ARITHMETIC OPERATORS

.*.	Multiply (numeric multiplication)
.+.	Plus (numeric addition)

Monolithic Memories PLEASM™ version 1.2D © copyright 1984 Monolithic Memories

PLEASM — PLE Assembler — provides the following options:

C Catalog	— Prints the PLEASM catalog of operations
E Echo Input	— Prints the PLE design specifications
T Truth Table	— Prints the entire truth table
B Brief Table	— Prints only used addresses in the truth table
H Hex Table	— Prints the truth table in HEX form
S Simulate	— Exercises the function table in the logic equations
I Intel Hex	— Generates INTEL HEX programming format
A ASCII Hex	— Generates ASCII HEX programming format
Q Quit	— Exits PLEASM

5

```
P5000                                          VINCENT COLI 10/03/82
BASIC GATES
MMI SANTA CLARA, CALIFORNIA
.ADD I0 I1 I2 I3 I4
.DAT O1 O2 O3 O4 O5 O6 O7 O8

O1 =  I0                                    ; BUFFER

O2 = /I0                                    ; INVERTER

O3 =  I0  *  I1  *  I2  *  I3  *  I4        ; AND GATE

O4 =  I0  +  I1  +  I2  +  I3  +  I4        ; OR GATE

O5 = /I0  + /I1  + /I2  + /I3  + /I4        ; NAND GATE

O6 = /I0  * /I1  * /I2  * /I3  * /I4        ; NOR GATE

O7 =  I0 :+: I1 :+: I2 :+: I3 :+: I4        ; EXCLUSIVE OR GATE

O8 =  I0 :*: I1 :*: I2 :*: I3 :*: I4        ; EXCLUSIVE NOR GATE

FUNCTION TABLE

I0 I1 I2 I3 I4 O1 O2 O3 O4 O5 O6 O7 O8

;INPUT   -  -  OUTPUTS  FROM  BASIC  GATES  -  -
;01234   BUF  INV  AND   OR   NAND   NOR   XOR   XNOR    COMMENTS
-------------------------------------------------------------------
 LLLLL    L    H    L    L    H      H     L     L      ALL ZEROS
 HHHHH    H    L    H    H    L      L     H     H      ALL ONES
 HLHLH    H    L    L    H    H      L     H     H      ODD CHECKERBOARD
 LHLHL    L    H    L    H    H      L     L     L      EVEN CHECKERBOARD
-------------------------------------------------------------------


DESCRIPTION

THIS EXAMPLE ILLUSTRATES THE USE OF PLEs TO IMPLEMENT THE BASIC GATES:
BUFFER, INVERTER, AND GATE, OR GATE, NAND GATE, NOR GATE, EXCLUSIVE OR
GATE, AND EXCLUSIVE NOR GATE.

NOTE ALSO THAT THREE-STATE OUTPUTS ARE PROVIDED WITH ONE ACTIVE LOW
OUTPUT ENABLE CONTROL (/E).

PLEASM GENERATES THE PROM TRUTH TABLE FROM THE LOGIC EQUATIONS AND
SIMULATES THE FUNCTION TABLE IN THE LOGIC EQUATIONS.
```

Figure 12a. PLE Design Specification. This is the source code for PLEASM. PLEASM generates the truth table and programming formats from the equations. PLEASM also exercises the Function Table in the equations and reports errors

```
BASIC GATES
.ADD I0 I1 I2 I3 I4
.DAT O1 O2 O3 O4 O5 O6 O7 O8

 ADD   A0 A1 A2 A3 A4   O1 O2 O3 O4 O5 O6 O7 O8
-------------------------------------------------
   0    L  L  L  L  L    L  H  L  L  H  H  L  L
   1    H  L  L  L  L    H  L  L  H  H  L  H  H
   2    L  H  L  L  L    L  H  L  H  H  L  H  H
   3    H  H  L  L  L    H  L  L  H  H  L  L  L
   4    L  L  H  L  L    L  H  L  H  H  L  H  H
   5    H  L  H  L  L    H  L  L  H  H  L  L  L
   6    L  H  H  L  L    L  H  L  H  H  L  L  L
   7    H  H  H  L  L    H  L  L  H  H  L  H  H
   8    L  L  L  H  L    L  H  L  H  H  L  H  H
   9    H  L  L  H  L    H  L  L  H  H  L  L  L
  10    L  H  L  H  L    L  H  L  H  H  L  L  L
  11    H  H  L  H  L    H  L  L  H  H  L  H  H
  12    L  L  H  H  L    L  H  L  H  H  L  L  L
  13    H  L  H  H  L    H  L  L  H  H  L  H  H
  14    L  H  H  H  L    L  H  L  H  H  L  H  H
  15    H  H  H  H  L    H  L  L  H  H  L  L  L
  16    L  L  L  L  H    L  H  L  H  H  L  H  H
  17    H  L  L  L  H    H  L  L  H  H  L  L  L
  18    L  H  L  L  H    L  H  L  H  H  L  L  L
  19    H  H  L  L  H    H  L  L  H  H  L  H  H
  20    L  L  H  L  H    L  H  L  H  H  L  L  L
  21    H  L  H  L  H    H  L  L  H  H  L  H  H
  22    L  H  H  L  H    L  H  L  H  H  L  H  H
  23    H  H  H  L  H    H  L  L  H  H  L  L  L
  24    L  L  L  H  H    L  H  L  H  H  L  L  L
  25    H  L  L  H  H    H  L  L  H  H  L  H  H
  26    L  H  L  H  H    L  H  L  H  H  L  H  H
  27    H  H  L  H  H    H  L  L  H  H  L  L  L
  28    L  L  H  H  H    L  H  L  H  H  L  H  H
  29    H  L  H  H  H    H  L  L  H  H  L  L  L
  30    L  H  H  H  H    L  H  L  H  H  L  L  L
  31    H  H  H  H  H    H  L  H  H  L  L  H  H
-------------------------------------------------

HEX CHECK SUM = 00F3C
```

Figure 12b. Truth Table. PLEASM generates this truth table which can be used for verifying your design

```
BASIC GATES
.ADD I0 I1 I2 I3 I4
.DAT O1 O2 O3 O4 O5 O6 O7 O8

 ADD      HEX ADDRESS      HEX DATA
-----------------------------------
   0          000             32
   1          001             D9
   2          002             DA
   3          003             19
   4          004             DA
   5          005             19
   6          006             1A
   7          007             D9
   8          008             DA
   9          009             19
  10          00A             1A
  11          00B             D9
  12          00C             1A
  13          00D             D9
  14          00E             DA
  15          00F             19
  16          010             DA
  17          011             19
  18          012             1A
  19          013             D9
  20          014             1A
  21          015             D9
  22          016             DA
  23          017             19
  24          018             1A
  25          019             D9
  26          01A             DA
  27          01B             19
  28          01C             DA
  29          01D             19
  30          01E             1A
  31          01F             CD
-----------------------------------

HEX CHECK SUM = 00F3C
```

Figure 12c. Hex Table. PLEASM generates this truth table in hexadecimal form for verification of locations in the PLE

```
32 D9 DA 19 DA 19 1A D9 DA 19 1A D9 1A D9 DA 19 .
DA 19 1A D9 1A D9 DA 19 1A D9 DA 19 DA 19 1A CD .

00F3C
```

Figure 12d. ASCII Hex Programming Format. PLEASM generates this ASCII Hex programming format with Hex check sum. Control characters are included so that the information can be down-loaded directly to a PROM programmer

```
:1000000032D9DA19DA191AD9DA191AD91AD9DA1940
:10001000DA191AD91AD9DA191AD9DA19DA191ACD54
:00000001FF
```

Figure 12e. Intel Hex Programming Format. PLEASM generates this Intel Hex Programming Format with a Hex check sum following every 16 bytes of data

PLE Family

Monolithic Memories carries a family of fast PROMs which can be used as Memory or PLE devices. Since the critical parameter for logic applications is speed, our series of fast PROMs have worst-case memory access times (or propagation delays) ranging from 15 ns for small PROMs to 40 ns for large PROMs. The Logic Symbols for four of the PLEs are given in Figure 13 and a summary of the PLE family is given below:

PLE Selection Guide

PART NUMBER	INPUTS	OUTPUTS	PRODUCT TERMS	OUTPUT REGISTERS	t_{PD} (ns) MAX*
PLE5P8	5	8	32		25
PLE5P8A	5	8	32		15
PLE8P4	8	4	256		30
PLE8P8	8	8	256		28
PLE9P4	9	4	512		35
PLE9P8	9	8	512		30
PLE10P4	10	4	1024		35
PLE10P8	10	8	1024		35†
PLE11P4	11	4	2048		35
PLE11P8	11	8	2048		35
PLE12P4	12	4	4096		35
PLE12P8	12	8	4096		40
PLE9R8	9	8	512	8	15
PLE10R8	10	8	1024	8	15
PLE11RA8	11	8	2048	8	15
PLE11RS8	11	8	2048	8	15

*Clock to output time for registered outputs

† Preliminary data.

NOTE: Commercial limits specified.

Acknowledgements

Several of the designs discussed in this paper were proposed by our good friend and colleague Ulrik Mueller, who is now studying Computer Science in his native country, Denmark, and our Monolithic Memories Pal Zahir Ebrahim. Special thanks also go to Ranjit Padmanabhan for writing the PLEASM simulator.

Summary

There are many interesting applications for high-speed PROMs used as PLEs. A software package called "PLEASM" is available as a development tool.

References

1. "PAL Programmable Array Logic Handbook", 3rd edition, J. Birkner, V. Coli, Monolithic Memories, Inc.
2. "Systems Design Handbook", Monolithic Memories, Inc.
3. "Bipolar LSI 1984 Databook", 5th edition, Monolithic Memories, Inc.
4. "PROMs and PLEs: An Application Perspective", Z. Ebrahim, Monolithic Memories Application Note AN-126.
5. "An Introduction to Arithmetic for Digital Designers", S. Waser, M.J. Flynn, Holt, Rinehart & Winston, N.Y., 1982.

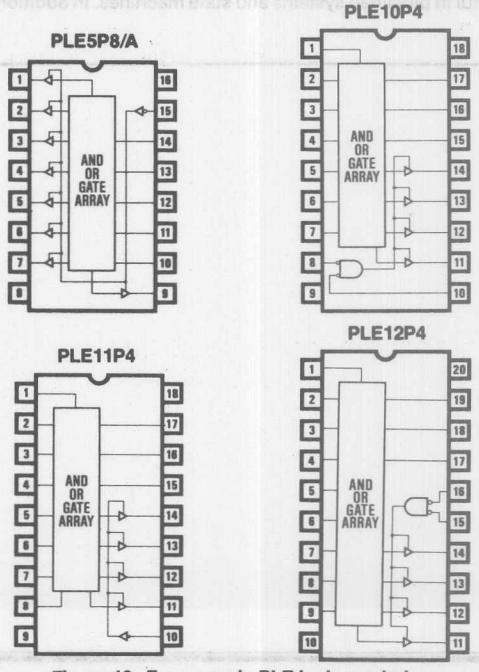


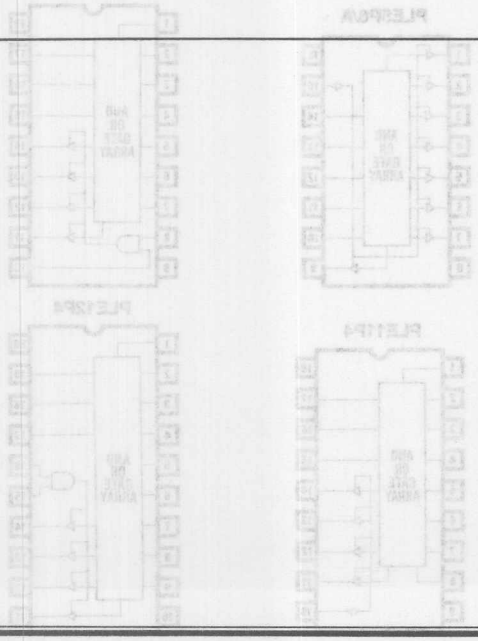
Figure 13. Four sample PLE logic symbols

High-Speed PROMs with On-Chip Registers and Diagnostics

Vincent J. Coli, Stephen M. Donovan and Frank Lee

Abstract

A family of High-Speed Registered and Diagnostic PROMs offer new savings for system designers. The Registered PROM family features on-chip "D"-type output registers which are useful in pipelined systems and state machines. In addition to output registers, the Diagnostic PROMs feature a Shadow Register which makes it easier for system designers to include diagnostics in microprogrammed systems. Architectures and applications for these devices are discussed in this paper.



TWX: 910-338-2376
2175 Mission College Boulevard, Santa Clara, CA 95050 Tel: (408) 970-9700 TWX: 910-338-2374

Monolithic Memories

High-Speed PROMs with On-Chip Registers and Diagnostics

Vincent J. Coli, Stephen M. Donovan and Frank Lee/Electro 84

A family of High-Speed Registered and Diagnostic PROMs offer new savings for system designers. The Registered PROM family features on-chip "D"-type output registers which are useful in pipelined systems and state machines. In addition to output registers, the Diagnostic PROMs feature a Shadow Register which makes it easier for system designers to include diagnostics in microprogrammed systems. Architectures and applications for these devices are discussed in this paper.

Architectures

In digital systems, it is natural to have a PROM followed by a register. This structure is particularly useful in microprogramming and state machine design. The Registered PROM family includes an on-chip Output Register as illustrated in Figure 1. By integrating these two building blocks into one chip, the following benefits are realized:

1. 2-to-1 chip count reduction
2. PC-Board space saving
3. Reduced power consumption
4. Eliminate the PROM output buffer and register input buffer and their associated delays.

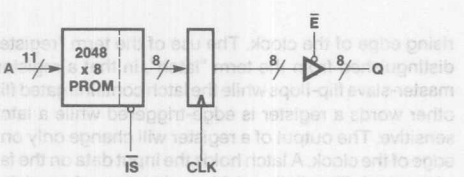


Figure 1. Registered PROM Block Diagram with Synchronous Initialization

In addition to the on-chip Output Register, the Diagnostic PROMs include extra circuitry to perform system level diagnostics, DOC™ (Diagnostic-On-Chip). Specifically, a buried Shadow Register with shifting capability and a 2:1 multiplexer are provided. A block diagram illustrating the Diagnostic PROM architecture is given in Figure 2.

Shadow register diagnostics allows observation and control of all points in a digital system by scanning through the Shadow Register. As a result, test vector generation is greatly simplified and a high degree of fault coverage can be easily obtained. A standalone 8-Bit Diagnostic Register (SN54/74S818 shown in Figure 3b) is also available. Several references are listed at the end of this paper which provide a detailed description of diagnostic architecture and how to use it, including Session 16 of this conference (see r4).

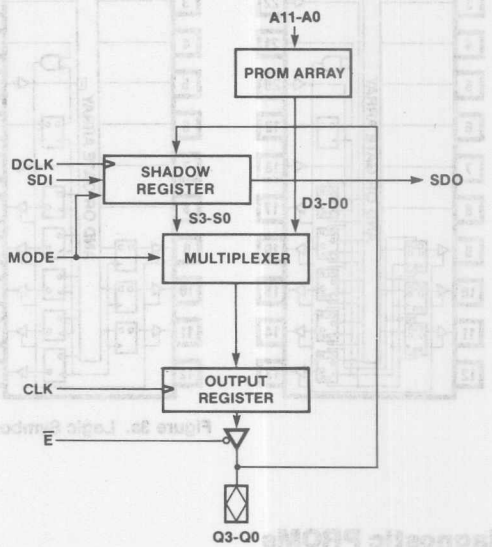


Figure 2. Diagnostic PROM Block Diagram

Product Families

Registered PROMs

The Registered PROMs are configured in 8-bit wide organizations with densities of 4K, 8K, and 16K. The following Registered PROMs are available:

53/63RA481 — 512 words x 8-bit memory with both synchronous and asynchronous three-state enables and preset and clear functions

53/63RS881 — 1024 words x 8-bit memory with both synchronous and asynchronous three-state enables and 16 synchronous initialization words

53/63RA1681 — 2048 words x 8-bit memory with asynchronous three-state enable and 16 synchronous initialization words

53/63RS1681 — 2048 words x 8-bit memory with synchronous three-state enable and 16 synchronous initialization words

5

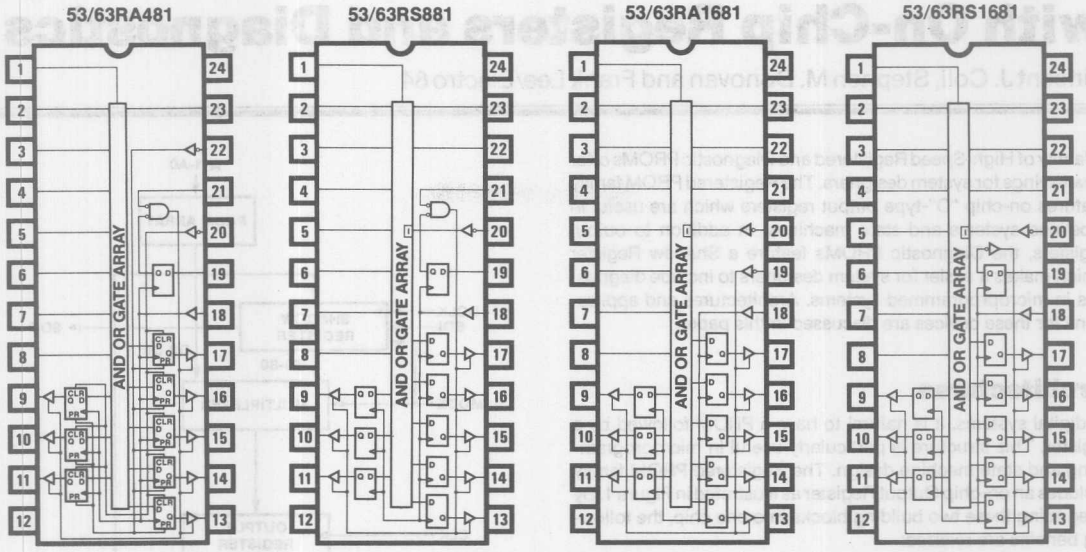


Figure 3a. Logic Symbols for the Registered PROM family

Diagnostic PROMs

The Diagnostic PROMs are configured in 4-bit wide organizations with densities of 4K, 8K and 16K. The following Diagnostic PROMs are available:

53/63DA441 — 1024 words x 4-bit memory with asynchronous initialization and two asynchronous three-state enables

53/63DA442 — 1024 words x 4-bit memory with both asynchronous and synchronous three-state enables

53/63DA841 — 2048 words x 4-bit memory with asynchronous initialization and asynchronous three-state enable

53/63D1641 — 4096 words x 4-bit memory with asynchronous three-state enable

53/63DA1643 — 4096 words x 4-bit memory with asynchronous initialization and totem-pole outputs

Both the Registered PROMs and Diagnostic PROMs are available in space-saving 24-pin SKINNYDIP® (0.3- inch wide) packages and are specified over both commercial and military temperature ranges.

Features

Edge Triggered Registers

Data from the PROM is loaded into the Output Register on the rising edge of the clock. The use of the term "register" is to be distinguished from the term "latch", in that a register contains master-slave flip-flops while the latch contains gated flip-flops. In other words a register is edge-triggered while a latch is level-sensitive. The output of a register will change only on the rising edge of the clock. A latch holds the input data on the falling edge of the clock. The distinguishing advantage of a register is that its output will only change on the rising edge of the clock, while a latch becomes transparent (output follows input) when the clock is High. As a result, system timing is simplified and faster microcycle times can be obtained.

Asynchronous Programmable Initialization

The Output Register can be loaded with a user-programmable initialization word. Each flip-flop in the Output Register may be individually programmed to either a HIGH state or a LOW state so that when the Initialize pin (I) is active (Low), the Output Register will now contain this initialization word. Note that the initialization operation will occur independent of a clock pulse. Also, this feature is a superset of a preset and clear function. Therefore programmable initialization can be used to generate any arbitrary microinstruction for system reset or interrupt. This feature is offered in several of the Diagnostic PROMs.

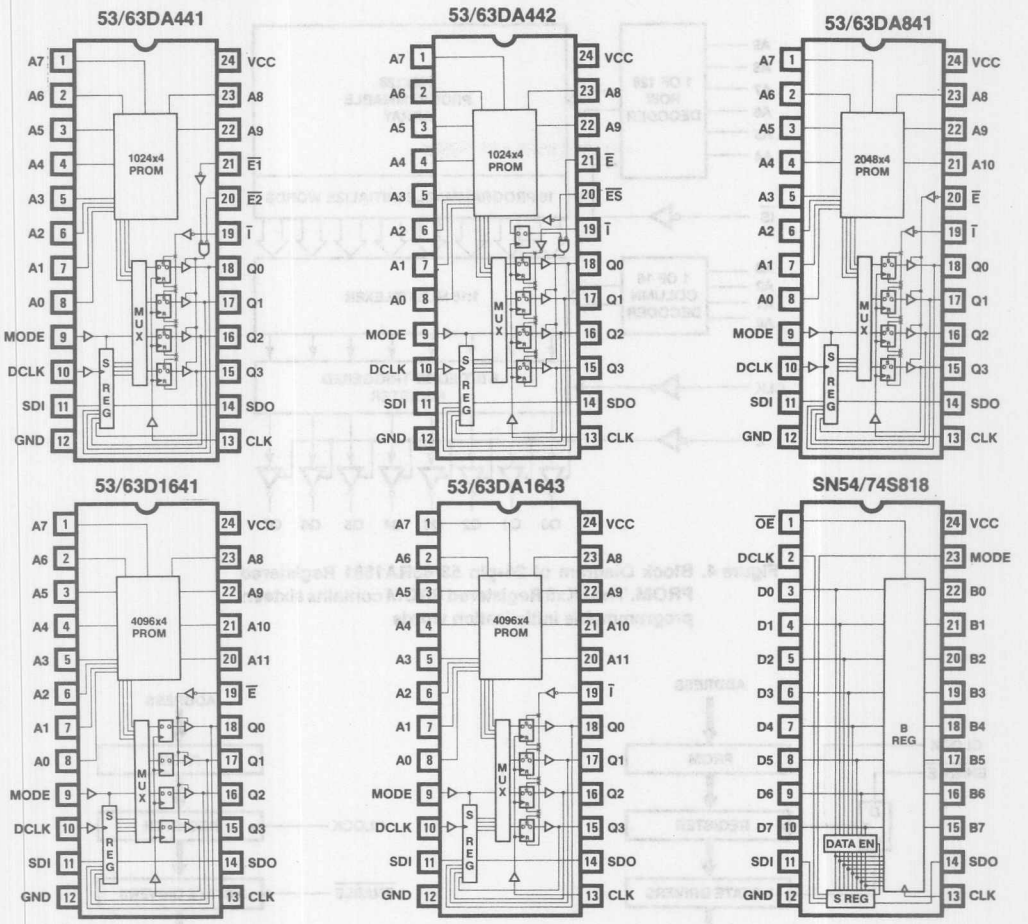


Figure 3b. Logic Symbols for the Diagnostic PROM family

Synchronous Programmable Initialization

This feature provides sixteen user-programmable synchronous initialization words. As illustrated in the Block Diagram (Figure 4), with the synchronous initialize pin ($\overline{IS}$) LOW, one of sixteen column words (A3-A0) will be loaded into the Output Register following the clock pulse and independent of the row addresses (A9-A4). This is useful for implementing a small ($\leq$16 word) reset or interrupt routine. With all $\overline{IS}$ column words (A3-A0) programmed to the same pattern, the $\overline{IS}$ function will be independent of both row and column addressing and may be used as a single pin control. This feature is offered in several of the Registered PROMs.

Three-State Drivers

The output of the register is buffered by three-state drivers which are compatible with low-power Schottky three-state bus standards. Thus VOL is 0.5 volts at IOL of 24 mA, VOH is 2.4 volts at IOH of -3.2 mA, and IOS minimum is guaranteed to be -20 mA. These hefty standards provide ample drive to meet the requirements of many bus standards.

Synchronous and Asynchronous Enables

Both synchronous and asynchronous output enable options are available. The synchronous output enable ($\overline{ES}$, see figure 5a), which is sampled on the rising edge of the clock, is used when more than one PROM is bused together to increase word length. In this case the enables effectively become the most significant address bits and, as such, must be registered just as data. Stated another way, when the clock goes high, the address is free to change , requiring enable information to be remembered somewhere. It is most appropriate to store the enable information. When the enable is not used, or when the outputs are to be gated onto some type of bus, the registered enable tends to get in the way. For this reason, the asynchronous output enable option ($\overline{E}$, see Figure 5b) is offered to allow direct control of the enable independent of the clock. For parts which have both synchronous and asynchronous output enables (see Figure 5c), outputs are enabled if, and only if, $\overline{ES}$ is LOW during the last rising edge of the clock and $\overline{E}$ is LOW.

5

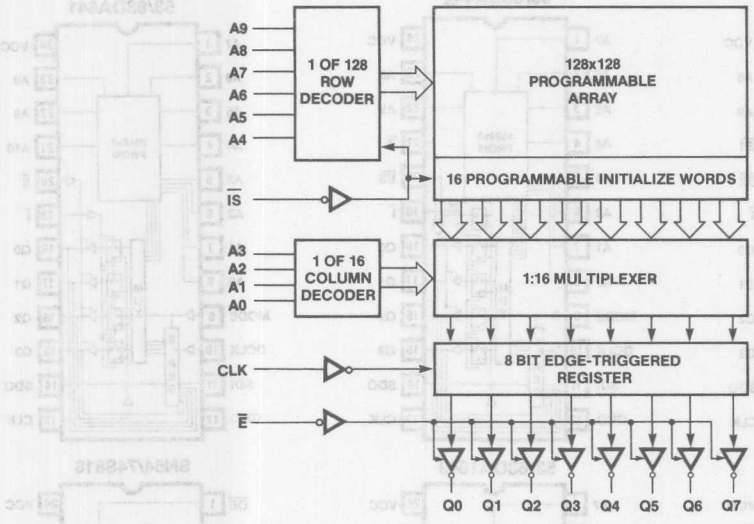


Figure 4. Block Diagram of 24-pin 53/63RA1681 Registered PROM. The 2Kx8 Registered PROM contains sixteen programmable initialization words

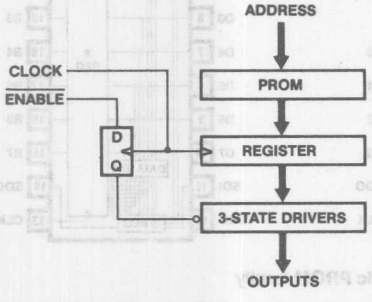


Figure 5a. Synchronous output enable ($\overline{ES}$)

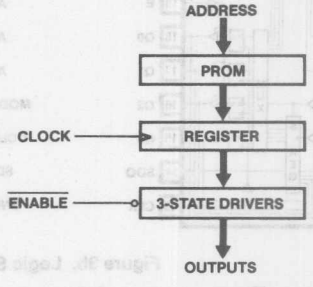


Figure 5b. Asynchronous output enable ($\overline{E}$)

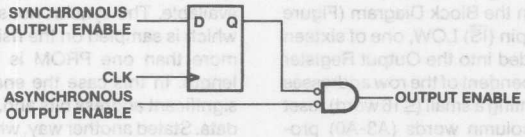


Figure 5c. Enabling of outputs for both synchronous ($\overline{ES}$) and asynchronous ($\overline{E}$) output enables

Application Areas

Microprogram Control Store

Microprogramming is the technique of using control programs stored in high-speed memory, such as bipolar PROMs, to instruct a digital system to perform various functions. A typical microprogram control store architecture is given in Figure 6.

The Microprogram Sequencer generates the addresses for the Microprogram Memory which stores the control program. The Microprogram register assures that all bits change simultaneously after the clock pulse and allows for pipelining instruction fetch and instruction execution. Some bits from the register are fed back to the sequencer while others are used for system control. This field of bits is called a Microword.

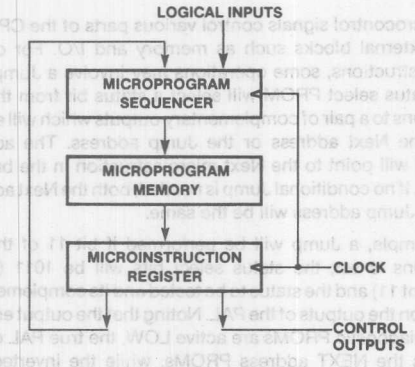


Figure 6. Typical Microprogram Control Store

Pipelined Systems

Pipelining is the art of designing digital systems such that delays associated with causal operations occur in parallel. A complex operation is divided into several smaller stages which are performed during clock cycles. Just as a widget traveling down an assembly line, each stage is operating on a piece of information which the previous stage operated on during the previous clock cycle. Maximum utilization of the hardware, which translates into maximum system performance, is achieved when the pipeline is full. The fall-through time for any piece of data through the system is the same (or even longer), but the number of pieces of data processed per unit time is greatly increased.

Clearly the benefit in pipelining microprogrammed systems is that instruction fetch and instruction execution times can be overlapped. Therefore the microcycle time is defined as the longer of either fetch or execution times, rather than the sum of both fetch and execution times, as illustrated in Figure 7.

Pipelining can also be used to obtain higher performance in data-intensive systems such as array processors where a large amount of data is coming in for processing without passing through the CPU. It is very inefficient to hold the next set of data until the previous data has propagated through all of the logic blocks in the system (Figure 8a). It is more efficient to pipeline the system and load new data after the previous data has been passed to the next block (Figure 8b).

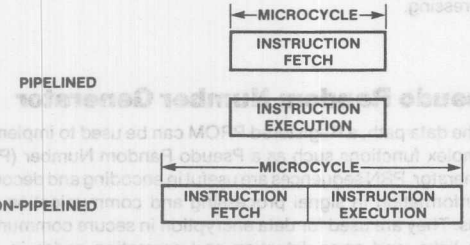


Figure 7. Length of Microcycle for pipelined and non-pipelined systems. Note that delays are overlapped in the pipelined system, while delays are summed in the non-pipelined system

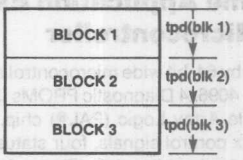


Figure 8a. An example of a nonpipelined (fall-through) approach to arithmetic operation

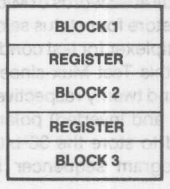


Figure 8b. Pipelined arithmetic operation. Note that t_{PD} = MAX [t_{PD} (blk 1), t_{PD} (blk 2), t_{PD} (blk 3)] + t_{PD} (reg)

Programmable Logic Elements (PLE™)

Since the inputs of PROMs are fully decoded and the outputs are definable for all possible input combinations, PROMs can be used as logic elements, replacing several levels of logic gates. PROMs are particularly useful for this application since the PROM provides a vast number of product terms (2^n, where n is the number of inputs) so that any transfer function can be implemented in a PROM with a sufficient number of inputs. The Output Register can be used to eliminate static hazards (glitches) which are normally unavoidable in PROMs. The Monolithic Memories trade name for high-speed PROMs used for logic is "PLE" (acronym for Programmable Logic Element). Monolithic Memories has developed a software tool called "PLEASM™" (PLE Assembler) to assist in designing and programming PROMs as PLEs. PLEASM is available for many computers and may be requested through the Monolithic Memories IdeaLogic Exchange. References r6, r7 and r8 offer an in-depth discussion of programmable logic applications for PROMs.

State Machines

A natural extension of using PROMs as logic elements is to use Registered PROMs as single-chip State Machines. In a classic state machine, the present state (or output) is a function of both the present inputs and the previous state. The combinatorial logic is implemented in the PROM array and the Output Register is used to store the state. One or more of the Registered PROM outputs are connected to address inputs in order to provide the state of the machine. The abundance of product terms in a PROM used to implement combinatorial logic translates into an unlimited combination of states. For example, a 2Kx8 Registered PROM can implement a 4-input, 8-output machine with any combination of 128 states. States and inputs can be traded off to provide a wide range of possible state machines. The programmable initialization feature is convenient to initialize the state machine.

5

A 4096-word by 64-bit wide microcontroller can be constructed using sixteen 4096x4 Diagnostic PROMs (53/63D1641) and one Programmable Array Logic (PAL®) chip. This controller supplies thirty-six control signals, four status select bits, and two addresses of twelve bits each (Figure 9) — one for the Next address and one for the Jump address.

In this design, three PROMs are used to store the Next address while an additional three PROMs are used to store the Jump address. Note that three 4-bit wide PROMs provide sufficient inputs to address the full 4096 words of Microprogram memory. One PROM is used to store four status select inputs to the PAL, which is used as a multiplexer for test conditions. A PAL16C1 or PAL20C1 is ideal for this Test Mux since these parts provide many inputs (sixteen and twenty respectively) and complementary output (both true and inverted) polarities. The remaining nine PROMs are used to store the 36-bit Microcontrol word. Note that a Microprogram sequencer is not used in this architecture.

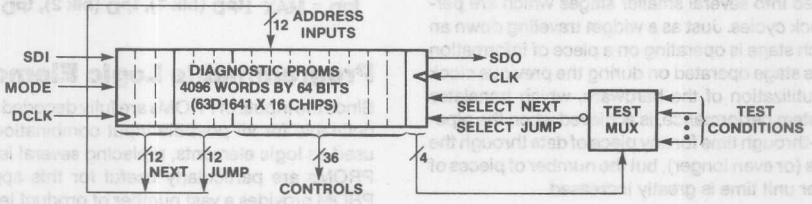


Figure 9. 64-bit Microcontroller using sixteen 4Kx4 Diagnostic PROMS and one PAL

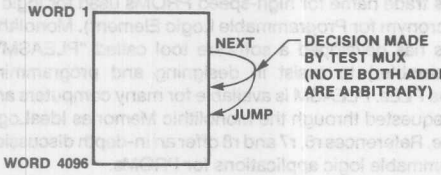


Figure 10. Microprogram Memory Map illustrating the Next address/Jump address decision made by the test mux

The decision cycle time is computed by the following equation:

$$f_{MAX} = \frac{1}{t_{su} + t_{CLK} + t_{PD} + t_{PXZ}}$$

where

t_{su} = address setup time for the diagnostic PROM
t_{CLK} = clock to output delay of the PROM
t_{PD} = propagation delay in the outside logic
t_{PXZ} = output enable/disable delay for the diagnostic PROM.

microinstructions, some operations may involve a Jump. The 4-bit status select PROM will select a status bit from the test conditions to a pair of complementary outputs which will enable either the Next address or the Jump address. The address enabled will point to the Next microinstruction in the bank of PROMs. If no conditional Jump is needed, both the Next address and the Jump address will be the same.

For example, a Jump will be performed if bit 11 of the test conditions is set; the status select bits will be 1011 (which represent 11) and the status to be tested and its complement will appear on the outputs of the PAL. Noting that the output enables of the Diagnostic PROMs are active LOW, the true PAL output controls the NEXT address PROMs, while the inverted PAL output controls the Jump address PROMs. If the test status is TRUE, the true PAL output disables the Next address PROMs while the inverted PAL output enables the Jump address PROMs. The reverse will occur when the test status is FALSE. This Next/-Jump decision is illustrated in Figure 10.

Note that the decision time can be decreased if the Next/Jump decision is made one clock cycle ahead and stored using a synchronous enable. This scheme will reduce the decision time by an amount equal to the propagation delay through the PAL Test Mux, but microcoding this system will become much more complex.

Fewer PROMs would be required if an even/odd Jump address scheme were used (such as only allowing Jumps to certain paragraphs), however this decreases the flexibility of PROM addressing.

Pseudo Random Number Generator

In the data path, a Registered PROM can be used to implement complex functions such as a Pseudo Random Number (PRN) Generator. PRN sequences are useful in encoding and decoding of information in signal processing and communication systems. They are used for data encryption in secure communication links, and error detection and correction codes in data communication systems. PRN sequences are also utilized as test vectors for testing digital systems and as reference white noise in many signal processing applications.

There are many techniques for generating PRN sequences. The most common technique is to use "n" stages of linear shift registers with feedback paths to determine a polynomial which characterizes a PRN sequence. Figure 11 illustrates a typical mechanism for generating PRN sequences.

The advantage of using a PROM (or PLE) for implementing PRN sequences is that any polynomial can be quickly customized in it. In data encryption systems where the code is frequently changed for protection from mischievous eavesdroppers, a PROM can be used to generate a new code each time or several codes can be implemented in the same PROM.

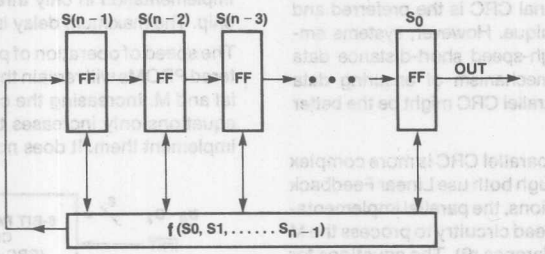


Figure 11. An "n" Stage Linear Feedback Shift Register (LFSR)). The PRN sequence generated is characterized by a polynomial of degree n. The feedback terms and logic functions determine its binary coefficients

5

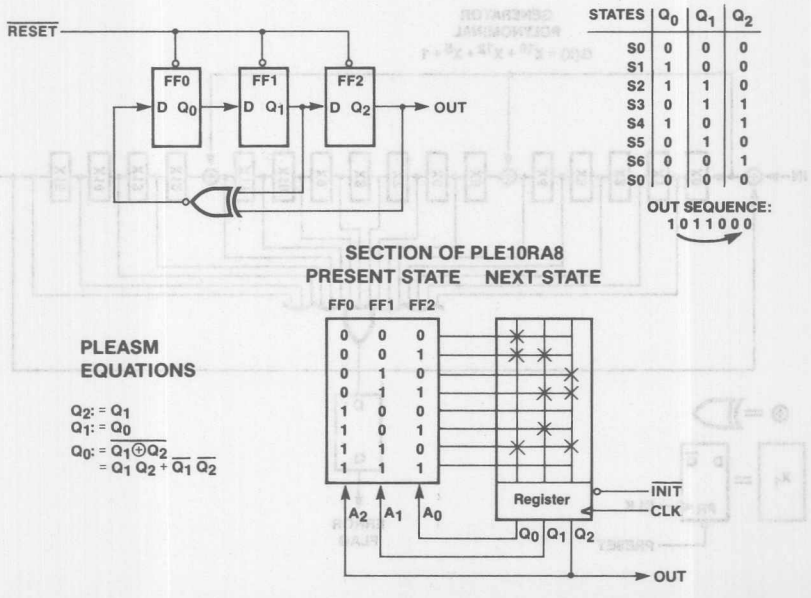


Figure 12. A 3-stage Pseudo Random Number Generator implemented in a Registered PROM (PLE)

An example of a PRN generator implemented in a Registered PROM is shown in Figure 12. A linear 2-input XOR function is used to generate a PRN sequence characterized by a polynomial of degree 3. The PRN sequence is of maximum length with period 7.

Cyclical Redundancy Check (CRC) is widely used for Error Detection in data communication. Both serial and parallel CRC can be performed depending on the nature of application. In serial data transfer on Local Area Networks, or between peripheral and main memory, serial CRC is the preferred and perhaps the most efficient technique. However, systems employing wide data buses for high-speed short-distance data transfer require a high-speed mechanism of ensuring data integrity. In these applications, parallel CRC might be the better alternative.

The implementation of an M-bit parallel CRC is more complex than its serial counterpart. Although both use Linear Feedback Shift Register (LFSR) configurations, the parallel implementation requires M-bit carry look-ahead circuitry to process the M data bits simultaneously (see reference r9). The equations for this carry look-ahead represent the output of each stage in the LFSR after every shift of an M-bit string of data. These equations contain a large number of XOR operations which make it very efficient to implement in a Registered PROM.

To illustrate with a practical example, Figure 14 shows the serial implementation of the CRC generator polynomial

$$G(X) = X^{16} + X^{12} + X^5 + 1$$

also called the CRC-CCITT standard. Figure 16 shows the 8-bit carry look-ahead equations for an 8-bit parallel CRC implementation of the same polynomial. These equations are derived in reference r9, where an implementation in four PAL devices is also shown with a maximum delay of 90 ns. Figure 15 shows an implementation in only three Registered PROMs and one SSI chip. The maximum delay is 50 ns.

The speed of operation of parallel CRC implemented in Registered PROMs will remain the same for any generator polynomial and M. Increasing the complexity of the carry look-ahead equations only increases the number of devices required to implement them. It does not increase the delay.

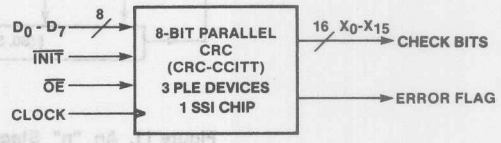


Figure 13. Block diagram of an 8-bit parallel CRC

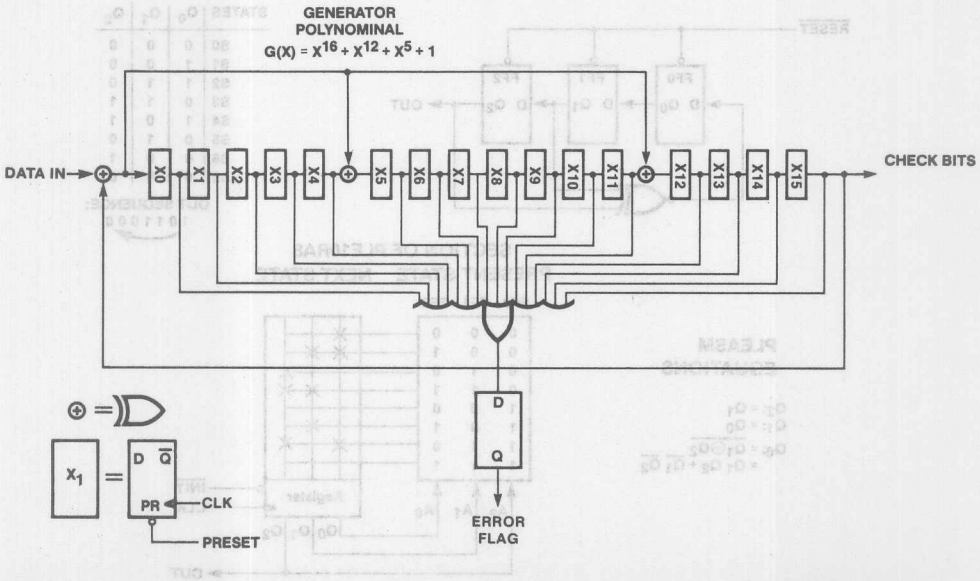


Figure 14. A 16-Bit Linear Feedback Shift Register (LFSR) Implementing a Serial CRC Generator

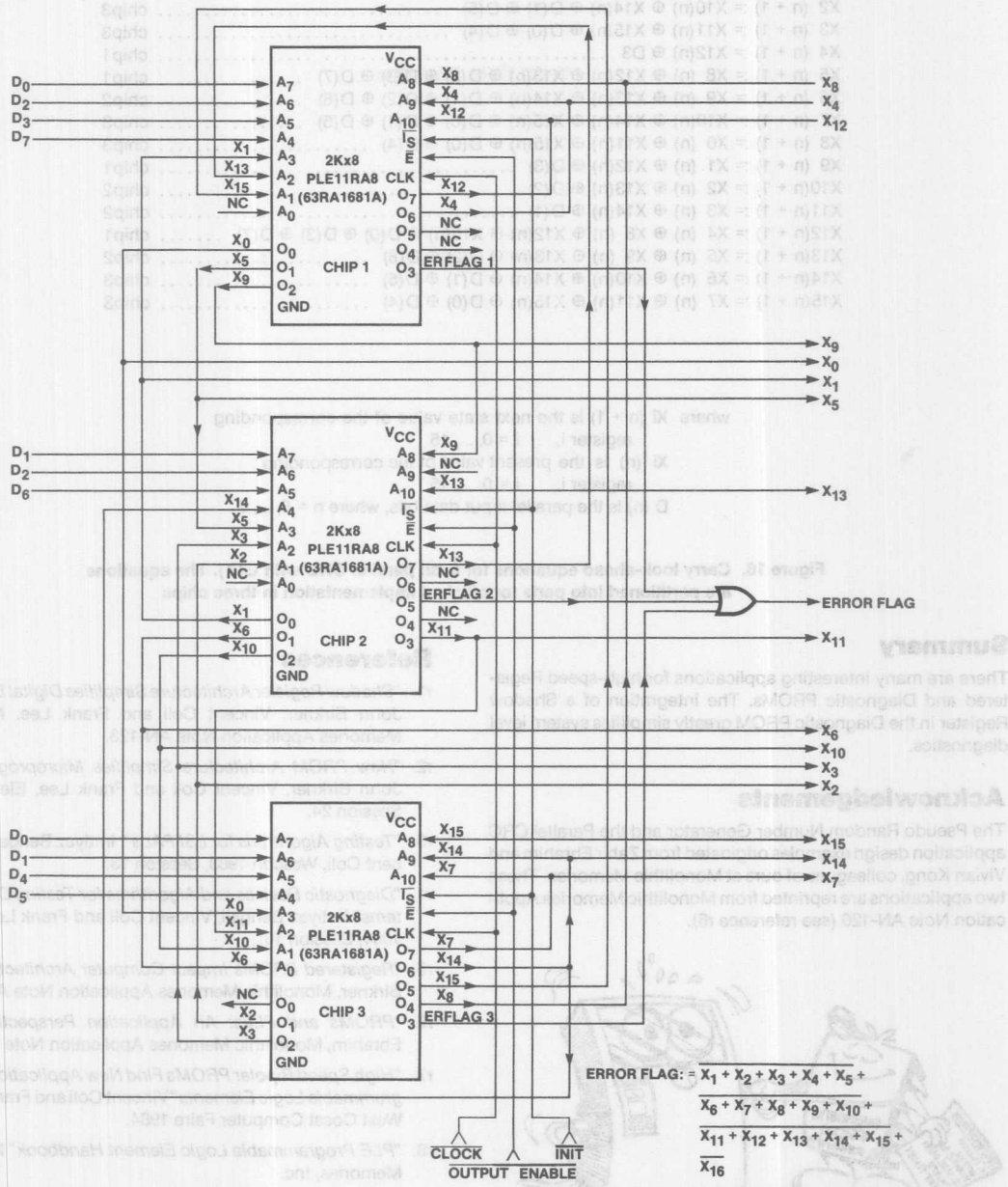


Figure 15. Diagram showing how to connect three Registered PROM (or PLE) devices together to implement 8-bit parallel CRC. The Error Flag is valid on the next clock pulse after all the data has been clocked in

X3 (n + 1) := X11(n) ⊕ X15(n) ⊕ D(0) ⊕ D(4) chip3
X4 (n + 1) := X12(n) ⊕ D3 ..... chip1
X5 (n + 1) := X8 (n) ⊕ X12(n) ⊕ X13(n) ⊕ D(2) ⊕ D(3) ⊕ D(7) chip1
X6 (n + 1) := X9 (n) ⊕ X13(n) ⊕ X14(n) ⊕ D(1) ⊕ D(2) ⊕ D(6) chip2
X7 (n + 1) := X10(n) ⊕ X14(n) ⊕ X15(n) ⊕ D(0) ⊕ D(1) ⊕ D(5) chip3
X8 (n + 1) := X0 (n) ⊕ X11(n) ⊕ X15(n) ⊕ D(0) ⊕ D(4) chip3
X9 (n + 1) := X1 (n) ⊕ X12(n) ⊕ D(3) chip1
X10(n + 1) := X2 (n) ⊕ X13(n) ⊕ D(2) chip2
X11(n + 1) := X3 (n) ⊕ X14(n) ⊕ D(1) chip2
X12(n + 1) := X4 (n) ⊕ X8 (n) ⊕ X12(n) ⊕ X15(n) ⊕ D(0) ⊕ D(3) ⊕ D(7) chip1
X13(n + 1) := X5 (n) ⊕ X9 (n) ⊕ X13(n) ⊕ D(2) ⊕ D(6) chip2
X14(n + 1) := X6 (n) ⊕ X10(n) ⊕ X14(n) ⊕ D(1) ⊕ D(5) chip3
X15(n + 1) := X7 (n) ⊕ X11(n) ⊕ X15(n) ⊕ D(0) ⊕ D(4) chip3

where Xi (n + 1) is the next state value of the corresponding register i, i = 0, ..., 15
Xi (n) is the present value of the corresponding register i, i = 0, ..., 15
D (n) is the parallel input data bits, where n = 0, ..., 7

Figure 16. Carry look-ahead equations for 8-bit parallel CRC with G(X). The equations are partitioned into parts for efficient implementation in three chips

Summary

There are many interesting applications for high-speed Registered and Diagnostic PROMs. The integration of a Shadow Register in the Diagnostic PROM greatly simplifies system level diagnostics.

Acknowledgements

The Pseudo Random Number Generator and the Parallel CRC application design examples originated from Zahir Ebrahim and Vivian Kong, colleagues of ours at Monolithic Memories. These two applications are reprinted from Monolithic Memories Application Note AN-126 (see reference r6).

"...THE DIAGNOSTIC PROMS AND DIAGNOSTIC REGISTERS HELP YOU TO ANALYZE YOUR SYSTEMS CONVENIENTLY !..."

References

r1. *"Shadow Register Architecture Simplifies Digital Diagnosis"* John Birkner, Vincent Coli and Frank Lee. Monolithic Memories Application Note AN-123.

r2. *"New PROM Architecture Simplifies Microprogramming"* John Birkner, Vincent Coli and Frank Lee, Electro 1983, Session 24.

r3. *"Testing Algorithms for LSI PALs"*, Imtiyaz Bengali and Vincent Coli, Wescon 1983, Session 13.

r4. *"Diagnostic Devices and Algorithms for Testing Digital Systems"* Imtiyaz Bengali, Vincent Coli and Frank Lee, Electro 1984, Session 16.

r5. *"Registered PROMs Impact Computer Architecture"* John Birkner, Monolithic Memories Application Note AN-107.

r6. *"PROMs and PLEs: An Application Perspective"* Zahir Ebrahim, Monolithic Memories Application Note AN-126.

r7. *"High Speed Bipolar PROMs Find New Applications as Programmable Logic Elements"* Vincent Coli and Frank Lee, 9th West Coast Computer Faire 1984.

r8. *"PLE Programmable Logic Element Handbook"* Monolithic Memories, Inc.

r9. *"Implementation of Serial/Parallel CRC Using PAL Devices"* Vivian Kong, Monolithic Memories Application Note AN-125.

Diagnostic Devices and Algorithms for Testing Digital Systems*

Imtiyaz M. Bengali, Vincent J. Coli and Frank Lee

Abstract

A new concept called Diagnostics On Chip™ (DOC™) was introduced in the industry recently. A series of new products with shadow register diagnostic capability is coming. These new products use this new concept and will provide a cost-effective solution to the issue of testability for digital systems.

5

* This paper is a slightly modified version of the paper by the same name which appeared in the Electro/84 Professional Program Session Record, Session 16 reprint, paper 16/1; 15-17 May 1984.

Diagnostic Devices and Algorithms For Testing Digital Systems

Imtiyaz M. Bengali, Vincent J. Coli, Frank Lee/Electro 84

A new concept called Diagnostics On Chip (DOC™) was introduced in the industry recently. A series of new products with shadow register diagnostic capability is coming. These new products use this new concept and will provide a cost effective solution to the issue of testability for digital systems.

Introduction

In developing a digital system, cost is a very sensitive issue. For the OEMs, cost itself can be categorized as R & D, manufacturing, marketing, testing, and maintenance costs, etc. The strategy is to reduce the overall cost for a system. Noting that marketing cost is about the same for a certain type of system and R & D cost is a one-time expense, it will be beneficial to put in diagnostic features to reduce the future expense in testing and maintenance.

If a large system goes down, it will not be practical to test all the chips individually. An alternative is to have built-in test circuits. If an error occurs, it can be located by running a test sequence through the system. It will definitely save a lot of time and expense compared to using tens or hundreds of man hours to debug the system manually.

Basics of Diagnostics

The test problem has two major facets:

1. Test generation.
2. Test verification.

Test generation is the process of determining the test sequence for a circuit which will demonstrate its correct operation. Test verification is to prove that the circuit works with the test vectors. Fault simulation has been the best technique of yielding a quantitative measure of test effectiveness. Test sequences are automatically generated and verified in the circuit after simulating a single "stuck-at-type" of fault in it. By observing the circuit outputs, faults can be detected and a quantitative measure of test effectiveness can be evaluated.

This technique is efficient to test combinatorial circuits, especially smaller circuits where the test sequence is trivial. For a more complex circuit, many techniques are available such as D-Algorithm, Compiled Code Boolean Simulation, Adaptive Random Test Generation, and other algorithms.

The techniques for combinatorial circuits are inefficient and ineffective for sequential circuits. As a first approximation, one can treat a sequential circuit as being purely combinatorial within each clock cycle and test it with the above techniques for a particular state. Every time the machine makes a transition to a new state, the test sequence is different. This is a very costly way of testing the circuit, especially if it has many states. Moreover, one should have the knowledge of initial state, illegal states, and sequences to bring the machine out from an illegal state into a known state. All these problems pertaining to testing of sequential circuits have given rise to the concept of "design for testability".

The key concepts are CONTROLLABILITY and OBSERVABILITY. Control and observation of a network are essential to implement its test procedure. Various designs for testability methods have evolved in the last five to six years. All of these methods have the same objective—to be able to control and observe critical points in a network. These techniques allow test generation problems to be completely reduced to the generation of test vectors for combinatorial logic.

For example, consider the case of the AND gate:

Figure 1. A simple 2-input AND gate

In order to test for a stuck-at-1 (sa1) fault, it is necessary to put 'A' to '0', 'B' to '1', and observe output 'C' for '0' or '1'. If '0' is observed at C, then the AND gate is good for sa1 fault; otherwise there is a fault. In order to fully test the AND gate, the following test vectors are to be exercised:

A	B	C	
0	1	0	Detect sa1
1	0	0	
1	1	1	Detect sa0

Table 1. A set of test vectors fully covering all stuck-at-faults of the AND gate in Figure 1

As the circuit becomes more complex, it is more difficult to control and observe every signal path. Thus, it becomes essential to give serious thought to the testability of the circuit through the design phase. One approach is to adopt structured design methodology. Ideally, this means that the design is totally synchronous with the system clock.

Most structured design practices are built upon the concept that if the values in all of the registers can be controlled to any specific value, and if they can be observed with a straightforward operation, then the test generation, and possibly the fault simulation task, can be reduced to doing test generation and fault simulation for a combinatorial network. A control signal can switch the memory elements from their normal mode of operation to a mode that makes them controllable and observable.

A simple but effective way to convert a sequential network to a combinatorial one is by breaking the feedback loop and inserting the test data in place of the sequential data in the feedback registers.

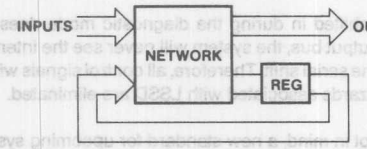


Figure 2a. A simplified representation of a sequential network

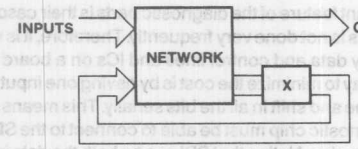


Figure 2b. The feedback path on the sequential network is broken in order to reduce the network to a pseudo-combinatorial one

There are many methods of design for testability practiced in the industry, like LSSD, ScanPath, Scan/Set, Random Access, and BILBO. All of these techniques require additional hardware, mostly shift registers, in order to input test sequences and to observe critical points in the circuit. It appears that additional cost in terms of special hardware has to be incurred for designing testability and structured design. But since the cost of hardware is declining, the trade-off is advantageous in the reduction of testing cost of bigger circuits.

Moreover, the circuit is well monitored and documented. Thus when the boards are in the field, and if there is a fault in a particular board, each block of the circuit can be monitored efficiently and the fault can be easily diagnosed, thus reducing maintenance cost in the future.

Previous Diagnostic Schemes

There are two basic methods to load in test vectors: parallel loading, and serial scanning.

Parallel loading of data in and out requires very wide input and output buses and is not worthwhile. Besides, it will not be effective to store and analyze the results. Built-in digital circuit observer (BIDCO) is a modified example of parallel loading of diagnostic data using a pseudorandom number generator to generate test vectors.

Serial scanning needs several clock cycles to load in or shift out the test results. It may take several minutes to run all the diagnostic vectors through the system. Considering the time needed to analyze the results and repair, the time taken to run the diagnostic vectors is insignificant. Examples of serial scan diagnostics techniques are level-sensitive scan design (LSSD) and Diagnostic-On-Chip (DOC). LSSD involves shifting of diagnostic data into latches, testing the system with that data and then shifting out the test result. DOC uses a buried register, called a shadow register, through which diagnostic data is shifted in and out.

Shadow Register—Why?

For the LSSD, outputs from the microcontrol store will contain some intermediate data when diagnostic microinstructions are shifted in and test results are shifted out. If several control signals are used to drive several ports on the same bus, it is possible that more than one port may be enabled at the same time by the intermediate data (as shown in Figure 4), thus creating a bus fight. The result may be hazardous to your system. Other hazards such as a disk crash are also possible. Designing with LSSD forced compromises in system design.

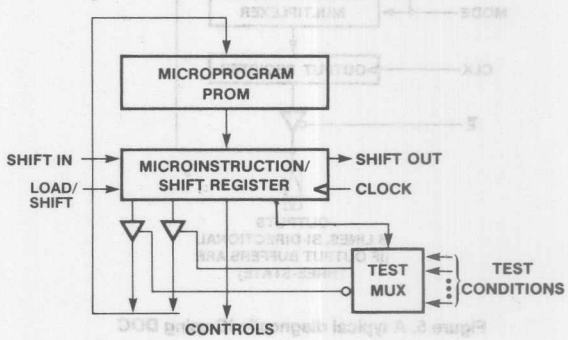


Figure 3. Serial scanning techniques simplify testing by serially shifting in test data and shifting out test result

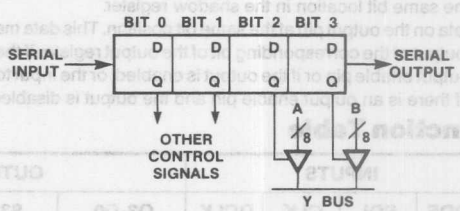


Figure 4. Potential bus fight may appear as Port A and Port B may both be enabled when test data or result is shifted through the register bits (also called three-state overlap)

If the diagnostic data is shifted into some buried registers which are not directly tied to the control lines, the above problem can be avoided. This is the concept of Diagnostic-On-Chip (DOC) which uses shadow register diagnostics.

An additional feature for a shadow register is to permit test vectors to be shifted in during normal execution, which means it is not necessary to hold up the system too long in order to perform diagnosis.

A shadow register is basically a buried register with shift capability (figure 5). There is also an output register whose outputs appear to

the output pin may be converted to an input pin. This feature is very important if the output is driving a bus and sampling of data on the bus is desired.

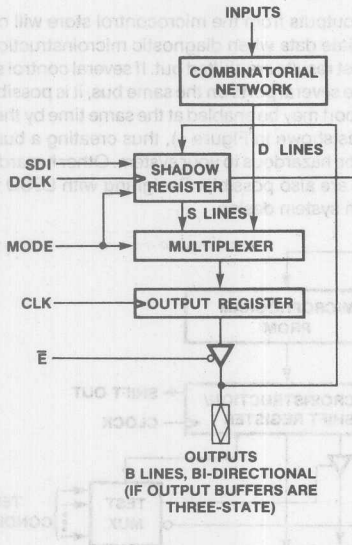


Figure 5. A typical diagnostic IC using DOC

The input to a bit of the shadow register is a multiplexer which can select data from one of three sources:
1) The less significant bit location in the shadow register (or SDI for the least significant bit). This operation is just a simple shift register.
2) The same bit location in the shadow register.
3) Data on the output pin at the same bit position. This data may be the output of the corresponding bit of the output register if there is no output enable pin or if the output is enabled, or the input to that pin if there is an output enable pin and the output is disabled.

Function Table

INPUTS				OUTPUTS			OPERATION
MODE	SDI	CLK	DCLK	Q3-Q0	S3-S0	SDO	
L	X	↑	*	Qn ← PROM	HOLD	S3	Load output register from PROM array
L	X	*	↑	HOLD	Sn ← Sn-1 S0 ← SDI	S3	Shift shadow register data
L	X	↑	↑	Qn ← PROM	Sn ← Sn-1 S0 ← SDI	S3	Load output register from PROM array while shifting shadow register data
H	X	↑	*	Qn ← Sn	HOLD	SDI	Load output register from shadow register
H	L	*	↑	HOLD	Sn ← Qn	SDI	Load shadow register from output bus
H	H	*	↑	HOLD	HOLD	SDI	No operation†

* Clock must be steady or falling. † Write back from shadow register to input bus (SN54/74S818 Diagnostic Register only).

Table 2. Operation of the diagnostic ICs in a digital system

2) The corresponding bit location in the shadow register.

Since the data shifted in during the diagnostic mode does not appear on the output bus, the system will never see the intermediate results of the serial shift. Therefore, all control signals will be valid and the hazards associated with LSSD are eliminated.

With this concept in mind, a new standard for upcoming system diagnostics can now be presented.

Cascadability of the Diagnostic ICs

One very significant feature of the diagnostic parts is their cascadability. Diagnostics is not done very frequently. Therefore, it is very costly to put many data and control lines and ICs on a board just for testing. One way to minimize the cost is by having one input line and one output line and shift in all the bits serially. This means that the SDO of a diagnostic chip must be able to connect to the SDI of another diagnostic chip. Noting that SDI can be both the data input or the control input, SDO must contain the most significant bit of the shadow register if SDI is the data input, and must pass the content of SDI if SDI is used as a control signal.

There is only one data input and one data output to the diagnostic parts. When serial data is shifted in or shifted out, data has to be passed from one diagnostic chip to another. Since SDI must be passed from chip to chip (if it is used for control), it is necessary for logic designers to make sure the fall-through time of SCI to the last chip and the setup time from SDI to DCLK are satisfied.

The Diagnostic IC Family

A family comprising 4K, 8K, and 16K Diagnostic PROMs (DPROM™) with 4-bit output organizations and 8-bit Diagnostic Register are available form Monolithic Memories. These devices are packaged in industry standard 24-pin SKINNYDIP® (0.3-inch wide) packages and are specified over both commercial and military temperature ranges.

The Diagnostic component series consists of the following products (see Figure 6 below for the Logic Symbols):

53/63DA441 — 1024 words x 4-bit memory with asynchronous initialization and two asynchronous three-state enables

53/63DA442 — 1024 words x 4-bit memory with asynchronous initialization and both asynchronous and synchronous three-state enables

53/63DA841 — 2048 words x 4-bit memory with asynchronous initialization and asynchronous three-state enable

53/63DA1641 — 4096 words x 4-bit memory with asynchronous three-state enable

53/63DA1643 — 4096 words x 4-bit memory with asynchronous initialization and totem-pole outputs

SN54/74S818 — 8-bit register with asynchonous three-state enable and write-back capability to the inputs, basically for loading of writeable control store (WCS). Even in the case of non-writeable control store, diagnostic registers should also be used in breaking the loops of the sequential system

The introduction of the DPROMs and diagnostic register results in a new standard for diagnostics. Noting that the diagnostic devices need controls over two independent registers and a multiplexer, a number of overhead pins are necessary. These overhead pins must be defined in a way that the diagnostic parts can be cascadable.

The diagnostic ICs need the following pins in addition to those used in a similar part without the diagnostic features:

1) Diagnostic Clock (DCLK)—The diagnostic clock is used to clock the shadow register.

2) MODE—This pin is used in selecting the data to the registers. For the output register, MODE = LOW indicates that the output register is being used as a normal register; MODE = HIGH indicates that the next state of the output register will be obtained from the shadow register. For the shadow register, MODE = LOW indicates serial data from SDI (see below) is shifted in every diagnostic clock; MODE = HIGH switches SDI from a data input to a control input. See below for details.

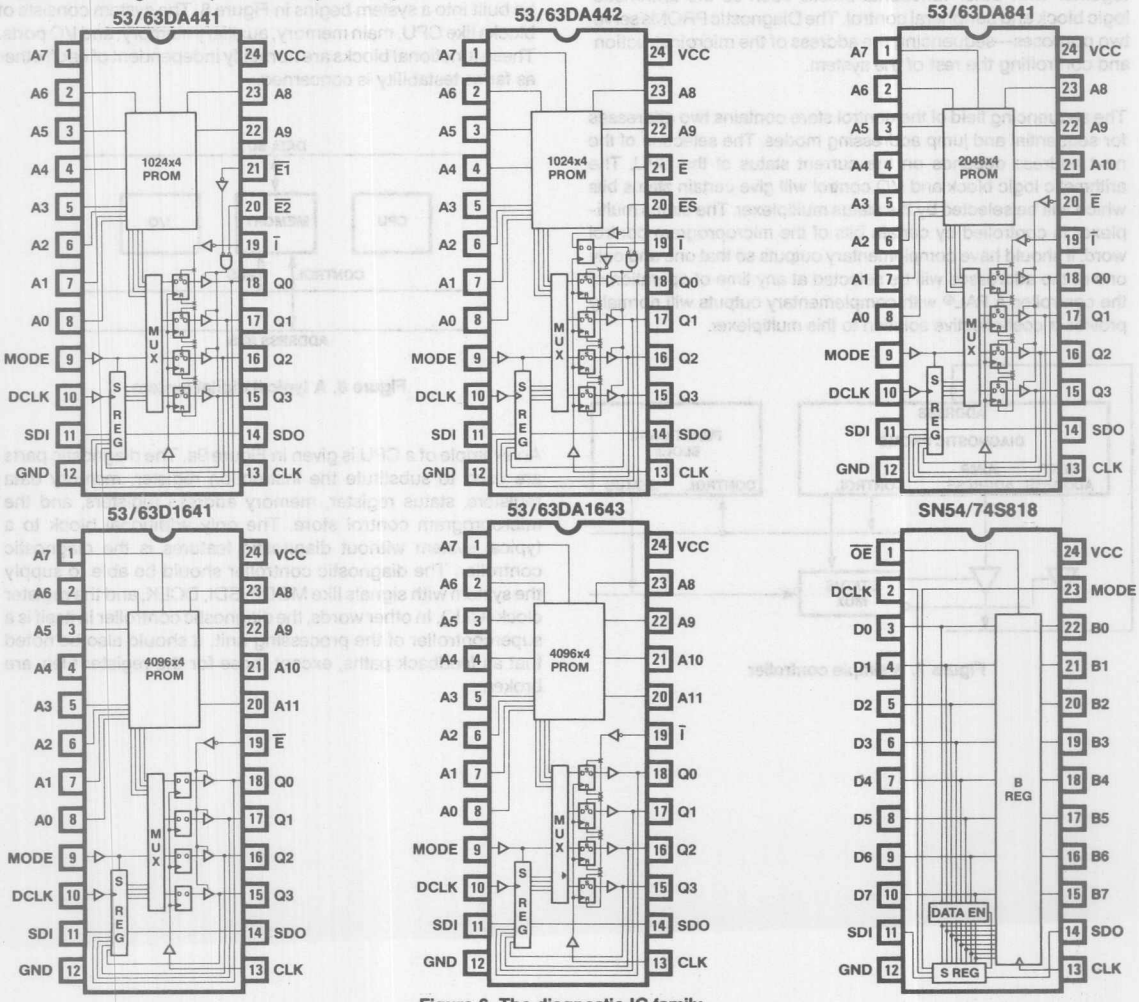


Figure 6. The diagnostic IC family

3) Serial Data In (SDI)—When MODE = LOW, this pin is used for shifting serial data in. When MODE = HIGH, SDI serves as a control pin. If MODE = HIGH and SDI = LOW, data from the output pins will be loaded to the shadow register on the next DCLK. MODE = HIGH and SDI = HIGH indicate a reserved operation for diagnostic PROMs, and is used for write-back for the diagnostic register.
4) Serial Data Out (SDO)—When MODE = LOW, this pin carries the shift-out bit of the shadow register. When MODE = HIGH, the SDI becomes a control pin and the control signal should be passed along if several diagnostic parts are connected together serially. So SDO should carry SDI along in this case

This standard is being used in designing all current and future diagnostic devices.

Some Applications Examples

A simple controller can be constructed using Diagnostic PROMs together with other functional blocks such as the arithmetic logic block and peripheral control. The Diagnostic PROMs serve two purposes—sequencing the address of the microinstruction and controlling the rest of the system.

The sequencing field of the control store contains two addresses for sequential and jump addressing modes. The selection of the next address depends on the current status of the CPU. The arithmetic logic block and I/O control will give certain status bits which will be selected by the status multiplexer. The status multiplexer is controlled by certain bits of the microprogram control word. It should have complementary outputs so that one and only one of the addresses will be selected at any time of operation of the controller. A PAL® with complementary outputs will normally provide a cost effective solution to this multiplexer.

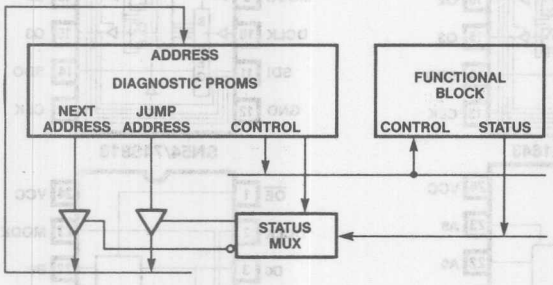


Figure 7. A simple controller

A continue statement can be implemented by having both addresses programmed to the next sequential address while an unconditional jump can be done by programming both addresses to the Jump address.

The control PROMs provide signals to control various functional blocks of the controller and other external blocks such as memory and I/O.

Since the other functional blocks such as the arithmetic logic and I/O control of the system also have sequential logic, it may be necessary to break the loops in those blocks so that diagnosis can be done on the whole system. The diagnostic registers can be incorporated in those blocks in places such as the registers (say, memory address registers, memory data registers, and instruction registers, etc.) The diagnostic data and result shifted into and out of the CPU can also be shifted through the diagnostic registers.

Another more detailed example of how the diagnostic parts can be built into a system begins in Figure 8. The system consists of blocks like CPU, main memory, auxiliary memory, and I/O ports. These functional blocks are normally independent of each other as far as testability is concerned.

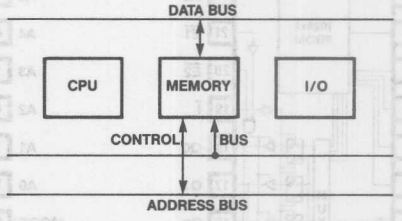


Figure 8. A typical digital system

An example of a CPU is given in Figure 9a. The diagnostic parts are used to substitute the instruction register, memory data registers, status register, memory address registers, and the microprogram control store. The only additional block to a typical system without diagnostic features is the diagnostic controller. The diagnostic controller should be able to supply the system with signals like MODE, SDI, DCLK, and the register clock (CLK). In other words, the diagnostic controller in itself is a supercontroller of the processing unit. It should also be noted that all feedback paths, except those for the register files, are broken.

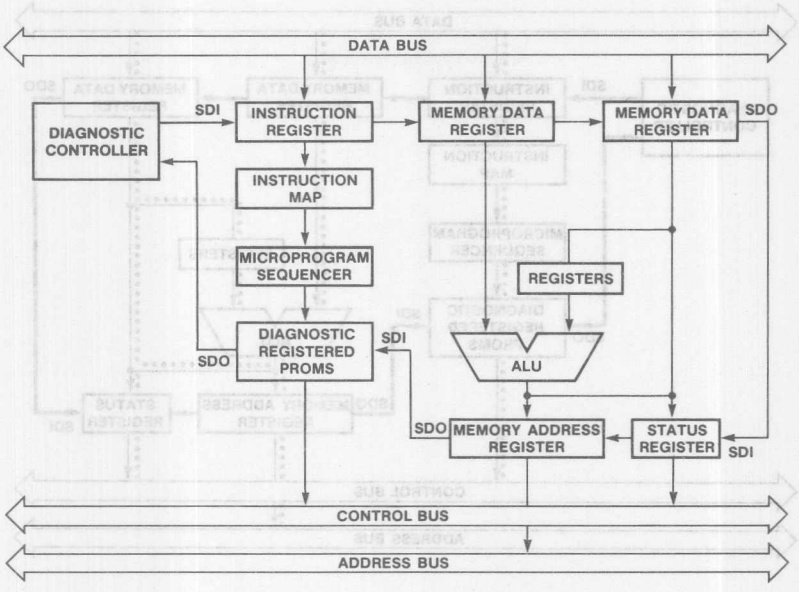


Figure 9a. A CPU using DPROMs and diagnostic registers

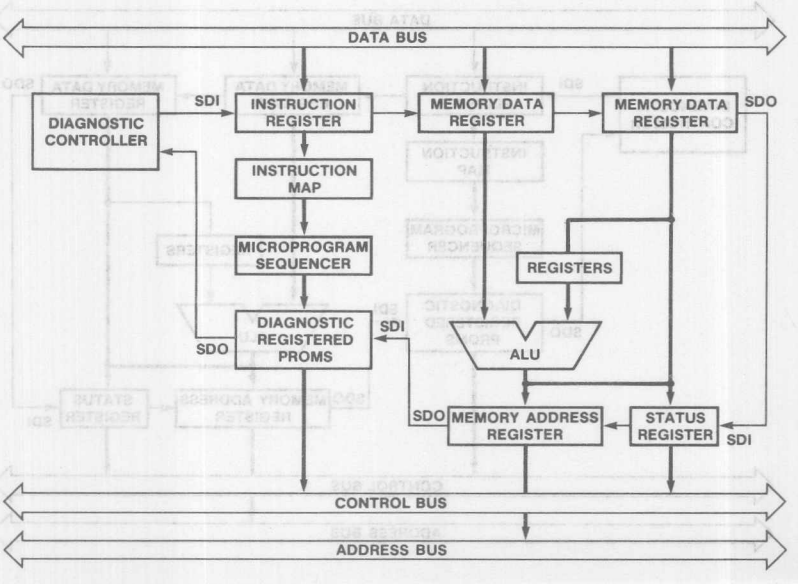


Figure 9b. Data flow during normal CPU operation

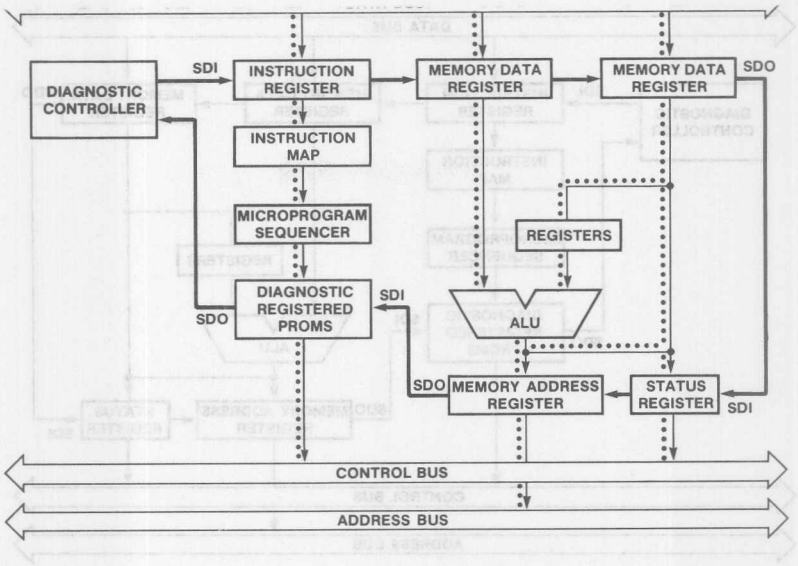


Figure 9c. Shifting on of test vector. Dotted lines represent data flow in the CPU if the loading of test vector is hidden in normal CPU operations

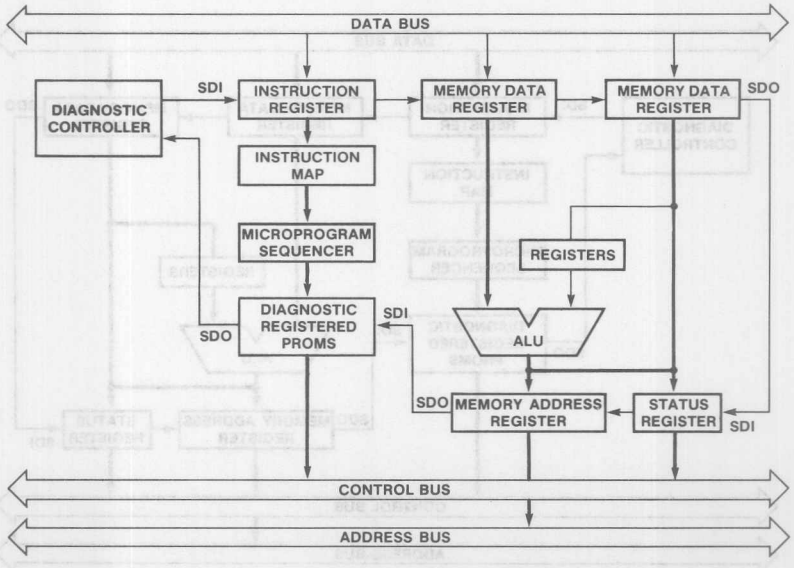


Figure 9d. After the last bit of the test data is shifted in, the output registers will be loaded with the test vector which will be used to test the system (Control the system)

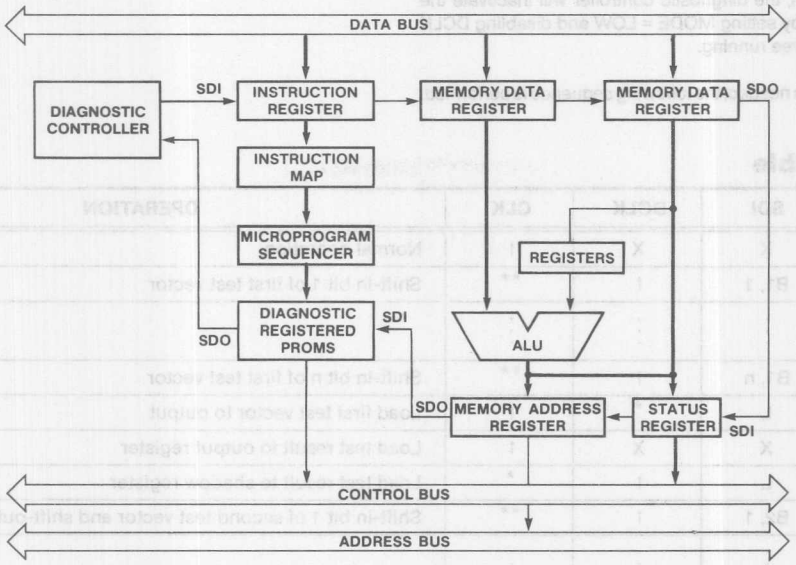


Figure 9e. The test result is then loaded back into the output register (Observe the system)

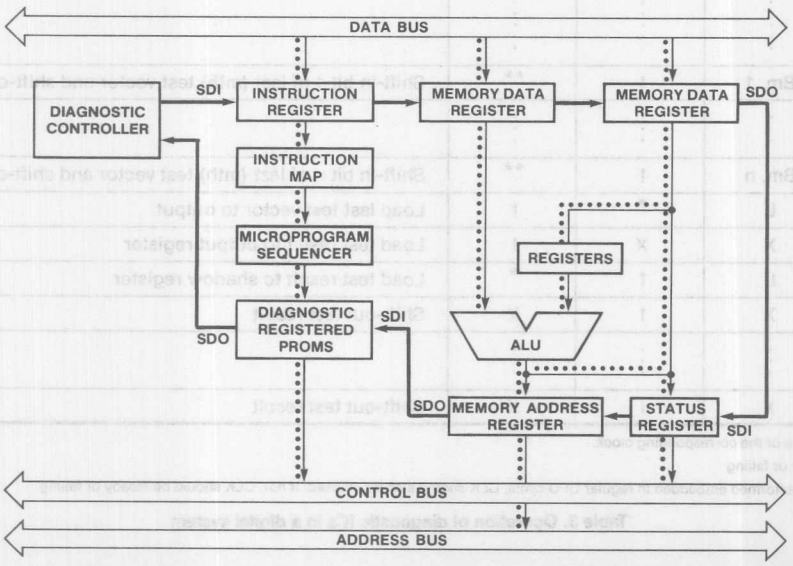


Figure 9f. The test result is shifted out of the same time the next test vector is shifted in. Again dotted lines represent data flow in the CPU if the loading of test vector is hidden in normal CPU operations

In normal operation, the diagnostic controller will inactivate the diagnostic feature by setting MODE = LOW and disabling DCLK and have the CLK free running.

When diagnostics is needed, the following sequence is performed:

Function Table

MODE	SDI	DCLK	CLK	OPERATION
L	X	X	↑	Normal operation
L	B1, 1	↑	**	Shift-in bit 1 of first test vector
⋮	⋮	⋮	⋮	
L	B1, n	↑	**	Shift-in bit n of first test vector
H	L	*	↑	Load first test vector to output
L	X	X	↑	Load test result to output register
H	L	↑	*	Load test result to shadow register
L	B2, 1	↑	**	Shift-in bit 1 of second test vector and shift-out test result
⋮	⋮	⋮	⋮	
L	B2, n	↑	**	Shift-in bit n of second test vector and shift-out test result
H	L	*	↑	Load second test vector to output
L	X	X	↑	Load test result to output register
H	L	↑	*	Load test result to shadow register
⋮	⋮	⋮	⋮	
L	Bm, 1	↑	**	Shift-in bit 1 of last (mth) test vector and shift-out test result
⋮	⋮	⋮	⋮	
L	Bm, n	↑	**	Shift-in bit n of last (mth) test vector and shift-out test result
H	L	*	↑	Load last test vector to output
L	X	X	↑	Load test result to output register
H	L	↑	*	Load test result to shadow register
L	X	↑	X	Shift-out test result
⋮	⋮	⋮	⋮	
L	X	↑	X	Shift-out test result

↑ Indicates a rising edge of the corresponding clock.

* Clock must be steady or falling

** If diagnosis is to be performed embedded in regular CPU cycle, CLK should also be clocked. If not, CLK should be steady or falling.

Table 3. Operation of diagnostic ICs in a digital system

A block diagram of a simple diagnostic controller is shown in Figure 10. The central control unit of this controller may be a disk-based unit or even another PROM. Since in normal operation MODE remains LOW and only CLK is active, it is possible to include a switch in Figure 10 so that the diagnostic controller will be inactive (see Figure 11).

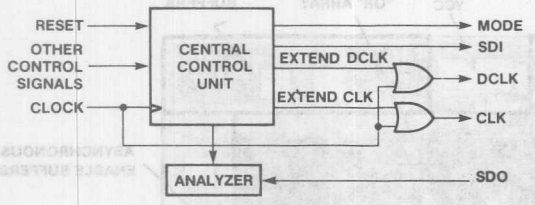


Figure 10. A block diagram of a diagnostic controller

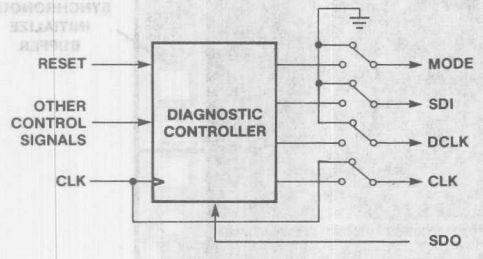


Figure 11. Including a switch to disconnect the diagnostic controller from the CPU

Some Final Thoughts

More complicated systems may have co-processors, DMA, I/O ports, etc., in addition to the CPU. A top-down approach will be very efficient in testing such systems by first locating the defective board, followed by the locating of the defective part in that board.

The diagnostic PROMs and registers can also be used in minicomputers, data storage devices, and peripherals.

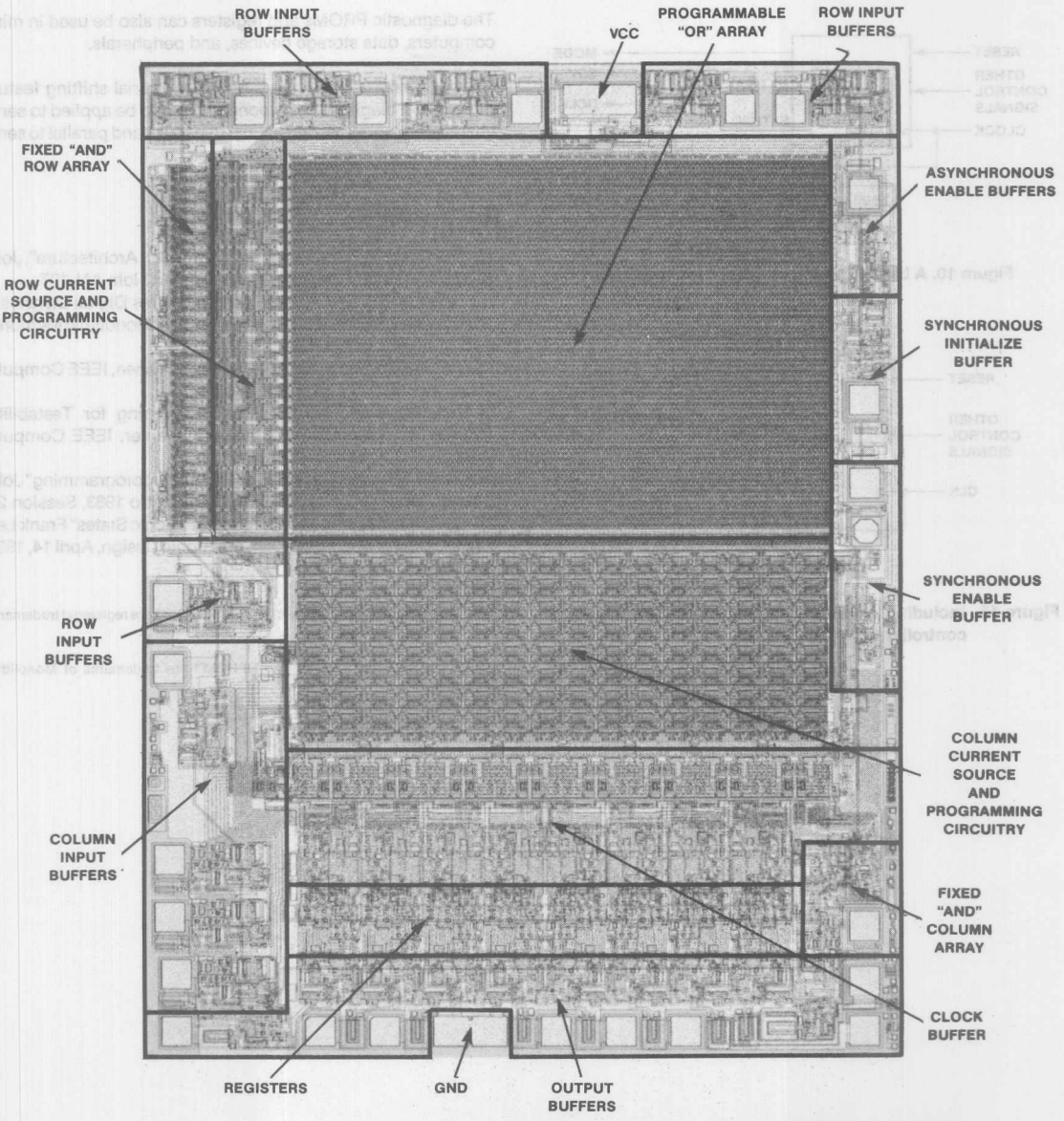
Besides being used for diagnostics, the serial shifting feature present in a diagnostic component can also be applied to serial character generators, other serial to parallel and parallel to serial converters, serial code generators, etc.

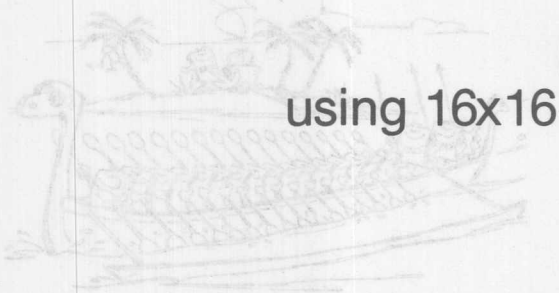
References

r1. "Registered PROMs Impact Computer Architecture" John Birkner, Monolithic Memories Application Note AN-107.
r2. "Shadow Register Architecture Simplifies Digital Diagnosis" John Birkner, Vincent Coli, and Frank Lee, Monolithic Memories Application Note AN-123.
r3. "Automated Testing of LSI" R.A. Rasmussen, IEEE Computer Magazine, March 1982, pp. 69-78.
r4. "Testing Logic Networks and Designing for Testability" Thomas W. Williams and Kenneth P. Parker, IEEE Computer Magazine, October 1979, pp. 9-21.
r5. "New PROM Architecture Simplifies Microprogramming" John Birkner, Vincent Coli, and Frank Lee, Electro 1983, Session 24.
r6. "On-Chip Circuitry Reveals System's Logic States" Frank Lee, Vincent Coli, and Warren Miller, Electronic Design, April 14, 1983, pp.119-124.

5

1Kx8 Registered PROM Metalization





Fast 64x64 Multiplication using 16x16 Flow-Through Multiplier and Wallace Trees*

Marvin Fox, Chuck Hastings and Suneel Rajpal

Abstract

The Monolithic Memories SN54/74S556 is a high-speed fully-parallel 16x16 multiplier and it provides the entire 32-bit product on a flowthrough basis from a single part. It is available in an 84-pin Leadless Chip Carrier (LCC) and 88-pin, pin-grid array packages. 8x8 40-pin array-multipliers such as the SN54/74S557/8 have been available for several years, however there is a large parts count for implementing longer wordlengths.

This paper describes the design philosophy and internal architecture of the 'S556 and applications for larger wordlength multiplications such as 32, 48, and 64 bits using these multipliers and high-speed PROMs and ALUs also available from Monolithic Memories.

The system advantages for using the 'S556 over the MPY-16H-class multipliers is also discussed; the main advantages being the availability of the entire product each cycle and the space savings on the board.

5

* This paper is a slightly updated version of the paper by the same name which appeared in the Northcon/83 Professional Program Session Record, Session 24 reprint, paper 24/2, 10-12 May 1983. A modified version subsequently also appeared in the Mini Micro West/83 Professional Program Session Record, Session 14, paper 14/2, 8-11 November 1983.

2175 Mission College Boulevard, Santa Clara, CA 95054 Tel: (408) 970-9700 TWX: 910-338-2374
TWX: 910-338-2376

Monolithic Memories

Summary

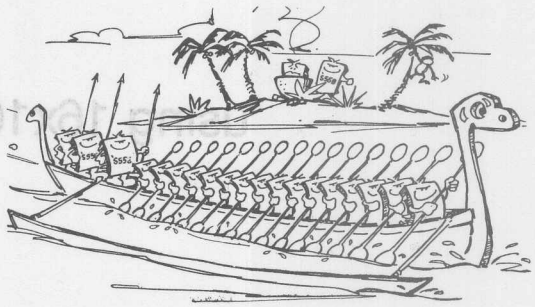
Multiplication is one basic digital-computer operation which can readily be speeded up by employing massive parallelism. "Cray multiplication" techniques, first used in large special-purpose computers a quarter of a century ago, are now commonplace in high-performance systems.

Essentially, in Cray Multiplication a full adder is placed in every position which would be occupied by a partial-product bit in a pencil-and-paper binary multiplication example (r1, r2). This technique may be applied within an LSI integrated circuit, in a system, or in both at once; it may or may not be modified by using "Booth-multiplication" approaches (r3, r4, r5).

8x8 40-pin Cray-multiplier integrated circuits have been available for several years, with a useful "flow-through" architecture. However the parts count for implementing full-blown Cray multiplication with practical scientific-computation word-lengths has been quite large. There have, of course, been several 16x16 Cray-multiplier 64-pin integrated circuits available; however, these have been unable, because of pin limitations, to furnish an entire 32-bit product in parallel. As a result, long-word-length multiplication cannot be performed economically on a flowthrough basis using these parts; some sort of clocking and multiplexing scheme is necessary to use them whenever the wordlength exceeds 16 bits, or else they must be duplicated outright.

Now there is a 16x16 Cray-multiplier part, the Monolithic Memories SN54/74S556, which provides the entire 32-bit product on a flow-through basis from a single part. The 'S556 has been designed to use the new 84-pin leadless-chip-carrier (LCC) and 88-pin pin-grid array packages, rather than compromising the architecture of the part because of the pin limitations (64 at most) of dual-in-line (DIP) packages.

This paper describes the design philosophy and internal architecture of the 'S556. It also shows how long-word-length multipliers may be built up from arrays of individual Cray-multiplier integrated circuits and programmable read-only memories (PROMs); the latter are used as "Wallace-tree" adders. Part-count and performance comparisons are made, for the representative word length of 64 bits, between implementations based on 64-pin 16x16 devices and implementations using 'S556s, in two different architectures; one which aims at lower cost and is a compromise between Cray multiplication and traditional shift-and-add multiplication.



"... MULTIPLICATION ... CAN READILY BE SPEEDED UP BY EMPLOYING MASSIVE PARALLELISM ..."

'S556 Architecture

The 'S556, shown in Figure 1, is a 16x16 Cray multiplier designed with an ultra-high-speed array of 256 adders, internally organized to the shift-and-add technique for multiplication (r1, r2). In place of the usual ripple-carry adders used in multiplier designs to sum up the final product bits, the 'S556 uses a carry-lookahead adder.

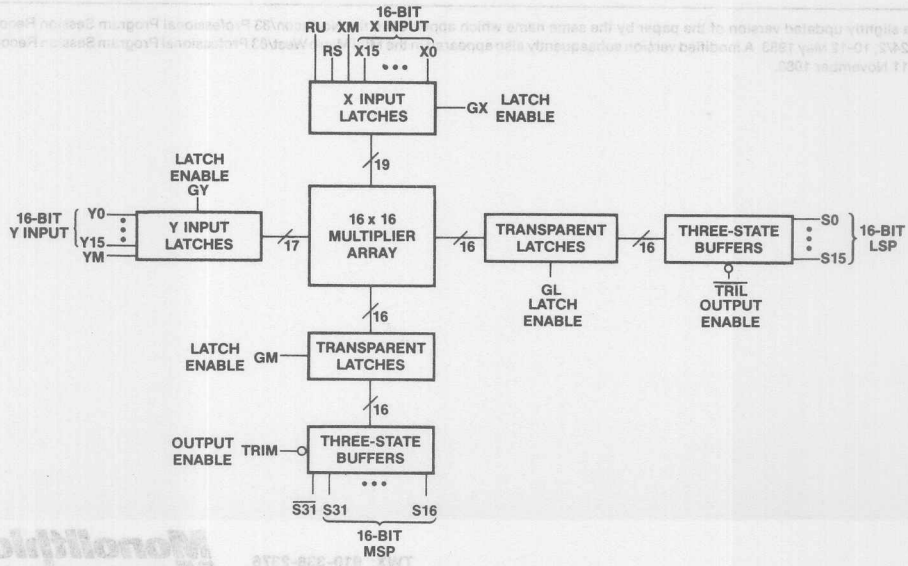


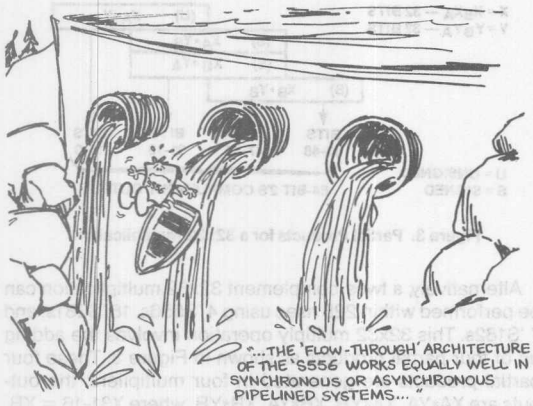
Figure 1. The 'S556 Architecture

The "flow-through" architecture of the 'S556 works equally well in synchronous or asynchronous pipelined systems. Latches are available to hold the input operands and the resulting double-length product, to increase the throughput rate in pipelined systems. If the designer does not wish to use these latches, they may be disabled, and the 'S556 then operates as a pure memoryless arithmetic network.

The 'S556 accepts operands in either unsigned or signed twos-complement form. When used in pipelined architectures, the 'S556 is capable of supplying 32-bit products at a 12.5 MHz repetitive throughput rate. The 'S556 has three-state outputs, controlled by the TRIL and TRIM control inputs.

Rounding-control input pins are provided on the 'S556 for rounding either unsigned or signed operands. Rounding is allowed in either of two binary positions, to support either "fractional-arithmetic" or "integer-arithmetic" positioning of a single-length rounded result.

The more traditional shift-and-add technique was chosen for the internal design of the 'S556 adder network because of the compactness, simplicity, and lower power requirement of this implementation. The Booth-algorithm approach, which groups the multiplier bits to effectively reduce the number of rows in the array, was considered (r3, r4, r5). However this approach also has penalties, in that it increases the width of each row from 16 to 18 bits, and the width of the final adder from 18 to 24 bits. Intrinsically, both the shift-and-add technique and the Booth-algorithm technique require 31 logic delays in the multiplier array using a ripple-carry final adder. At this point, the use of a carry-lookahead adder structure results in major speed improvements.

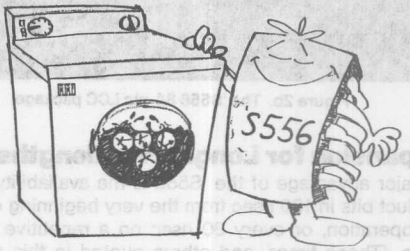


Here again there are tradeoffs. In MSI bipolar circuits, carry-lookahead parts are reasonable to construct with scanning widths of up to 4 bits, with a carry-out available (r5). Beyond that, the circuit gets bulky and power-hungry. Parallel "banking" of 4-bit-adder groups may be used to extend this limit, but here again 4 to 5 banks is as far as this approach can be reasonably pushed. With parallel banking the 24-bit adder required by the Booth-algorithm technique can be implemented using 6 banks of 4-bit adders; this exceeds the limit of 4 to 5 banks. This shows that the Booth-algorithm implementation requires fewer horizontal rows of adders, which translates to shorter propagation delays as compared to shift-and-add technique; however the final adder in the Booth-algorithm implementation is slower than the final adder in the shift-and-add technique implementation.

The 'S556 internal design uses an Emitter-Coupled-Logic (ECL) circuit implementation, based on Monolithic Memories' new washed-emitter process. ECL was chosen here over TTL and Emitter-Follower Logic, both of which have been used in previous Monolithic Memories Cray-multiplier designs (r3, r4). Here, ECL also turns out to have the most compact circuit-layout form, requiring 82 square mils of chip surface area per full adder. Emitter Function Logic (EFL) was chosen for one portion of the design, the carry-lookahead tree, because it interfaces easily with single-ended ECL outputs. All latches are implemented in ECL, to interface easily with the TTL/ECL buffers at the inputs and the ECL/TTL buffers at the outputs. The input latches introduce one ECL delay, but there is zero additional delay at the outputs as the output latches are incorporated right into the ECL/TTL translators.

The 'S556 is a universal multiplier aimed at a flow-through-type-processor architecture. Latches are used since registers cannot implement a flow-through architecture directly.

To be sure, the currently-available 16x16 multipliers from TRW and AMD, which use 64-pin dual-in-line packages do have a feed-through capability on the output registers. This capability allows latch-like transparency on the output registers, but nowhere else, since the parts are pin-limited and input and output data must in some cases share the same pins. Such an implementation consumes considerably more chip area and power than a purely latch design.



"...THE 'S556 INTERNAL DESIGN USES MONOLITHIC MEMORIES' NEW WASHED-EMITTER PROCESS..."

Many users who wish to use registers to achieve pipelined operation can find ways to do so using the 'S556s' internal latches. Usually pipelining can be achieved by choosing the proper phasing and pulse width of the latch gate-control signals without resorting to using external registers. Of course, external registers may be used when absolutely necessary.

The 'S556 will be supplied in an 84-pin Leadless Chip Carrier (LCC), and also in an 88-pin pin-grid-array package, with an integral heat sink. Both Commercial and Military grade parts will be available. The pinout is shown in Figure 2a. A photograph of the 84-pin LCC package is shown in Figure 2b.

Figure 2a. The 'S556 Pinout Diagram

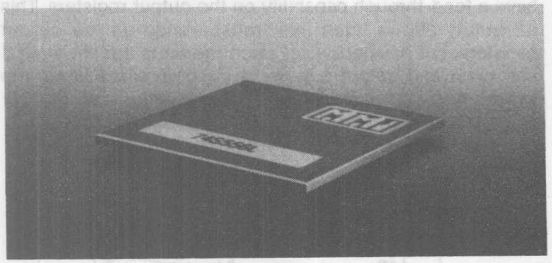


Figure 2b. The 'S556 84-pin LCC package

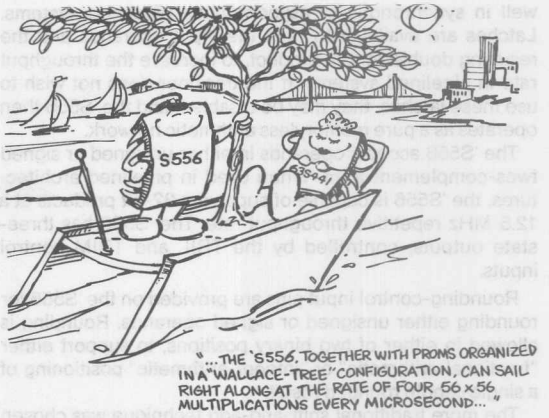
Expansion for Longer Wordlengths

A major advantage of the 'S556 is the availability of all 32 product bits in 100 nsec from the very beginning of a multiply operation, or every 80 nsec on a repetitive pipelined basis. (These times, and others quoted in this paper, are worst-case rather than typical.) Thus, the 'S556 is especially suited for longer-wordlength arithmetic units.

Other commercially-available multipliers, of the TRW MPY-16H class, are packaged in 64-pin 900-mil DIPs, which require a circuit board area of approximately 1" x 3.25". Moreover, these parts operate more slowly in expanded configurations, as the most-significant half and the least-significant half of the 32-bit double-length product must be obtained on two successive clock cycles.

Totally-Parallel 32-bit Multiplier

The 'S556, together with PROMs organized in a "Wallace-Tree" configuration, can sail along at the rate of four 56x56 multiplications every microsecond. An unsigned 32-bit multiplication can be performed using 4 'S556 multipliers, 11 63S481A PROMs used as "Wallace-Tree adders" (r1), and 16 'S381 and 5 'S182 used to form a 64-bit adder. The multipliers supply the partial products which are positioned as shown



in Figure 3. The difference is that only unsigned operands are used, and only positive partial products are added. The three rows of partial products which overlap are added by using PROMs which "compress" these three rows to 2 rows, which are then added in the 64-bit adder. The compression technique is discussed in greater detail in the description, later on, of the 64-bit multiply operation. Using the above configuration, an unsigned 32x32 multiply operation can be performed in less than 175 nsec worst-case allowing for a 75-nsec 'S556 multiplier delay, a 30-nsec 63S481A PROM delay and a 64-nsec 64-bit adder delay.

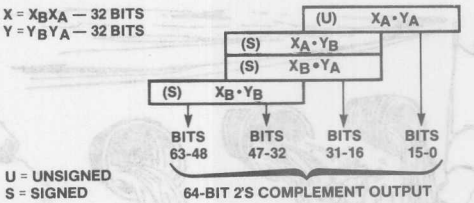
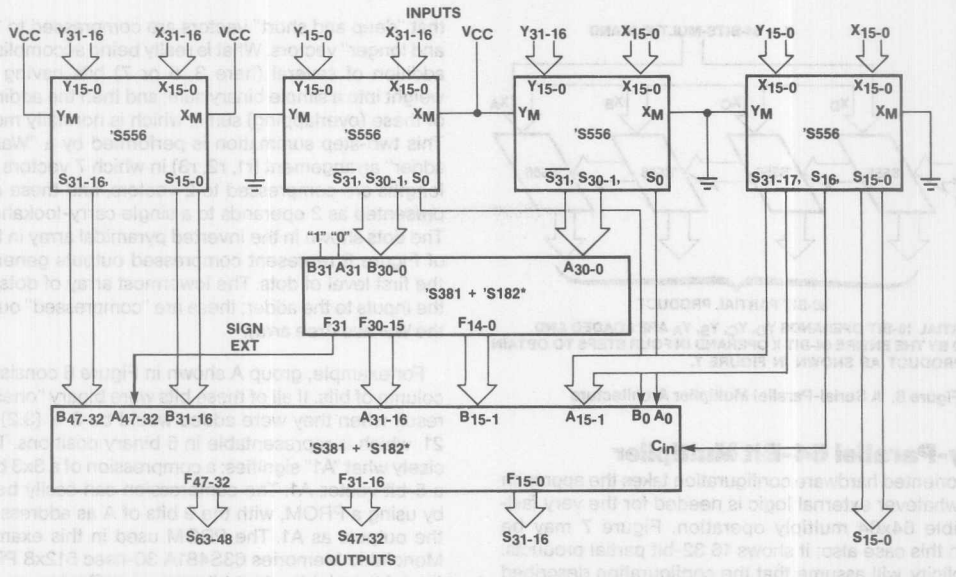


Figure 3. Partial Products for a 32x32 Multiplication

Alternatively, a twos-complement 32x32 multiplication can be performed within 228 nsec using 4 'S556s, 18 'S381s, and 7 'S182s. This 32x32 multiply operation involves the adding up of four partial products as shown in Figure 3. These four partial products are generated in four multipliers; the outputs are XA*YA, XA*YB, XB*YA, XB*YB, where X31-16 = XB, X15-0 = XA, Y31-16 = XB, Y15-0 = XA.

The implementation of this twos-complement 32x32 multiplier is shown in Figure 4. The outputs of the 16x16 multipliers are connected to two levels of adders to give a 64-bit product. The first level of adders is needed to add the two central partial products of Figure 2, XA*YB and XB*YA. Notice the technique which is used to generate the "sign extension" or the most-significant sum bit of the first level of adders. The 'S556 provides as a direct output the complement of the most-significant product bit; having this signal immediately speeds up the sign-extension computation, and reduces the external parts count.



* THESE ARE ADDER BLOCKS USING THE 'S381, A 4-BIT ALU FUNCTION GENERATOR, TO PERFORM A HIGH SPEED ADD OPERATION. THE 'S182 IA A LOOK-AHEAD CARRY GENERATOR AND IT REDUCES THE PROPAGATION DELAY. ALL THE ABOVE PARTS ARE AVAILABLE FROM MONOLITHIC MEMORIES INCORPORATED.

TOTAL MULTIPLY TIME = MULTIPLIER DELAY + ADDER LEVEL 1 DELAY + ADDER LEVEL 2 DELAY = 75 + 64 + 64 = 203 nsec

Figure 4. Implementation of the 32x32 Multiplier

For example, the inputs to the adder in the most significant position are the $\overline{S31}$ outputs from the two central multipliers. The sign extension of the addition of XA*YB and XB*YA is defined as

SIGN EXT = $\overline{A.B. + A.C + B.C}$, where

A is the most-significant bit of the term XA*YB;

B is the most-significant bit of the term XB*YA; and

C is the carry-in to the most-significant bits of XA*YB and XB*YA, in the adder.

The sign extension can be computed as the negation of the carry-out term of three terms, A, B, and C. This term corresponds to the negative of the carry-out of the bit position just one place to the right of the most-significant bit position of the first level of adders. The negative of the carry-out can be generated by presenting a carry-out and a binary "one" to the most significant bit of the adder. The generated sum bit then corresponds to the negation of the carry-out of the previous stage, which is the sign extension required to be added to the 16 most-significant bits of the XB*YB partial product term.

The second level of adders, which performs a 40-bit add function, is fairly straightforward. These adders can be implemented using 'S381 four-bit ALUs and 'S182 carry-bypasses ("carry-lookahead generators") which are available from Monolithic Memories, Inc. and from other vendors.

Other configurations such as 48x48 multipliers can be designed using the same methodology. Figure 5 shows the alignment of the partial products from 9 'S556s for the 48x48 case.

Serial-Parallel Multiplier

In applications where speed can be sacrificed, it is possible

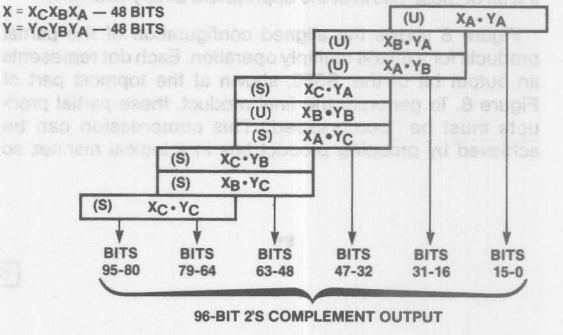
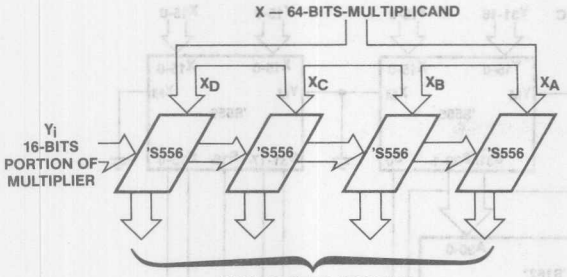


Figure 5. Partial Products for a 48x48 Multiplication

to implement an alternative solution using fewer multipliers, at some penalty in speed, but still with a very significant speed gain over other methods of multiplication. Figure 6 shows a plausible method of performing a 64x64 multiply operation, in four cycles. Each cycle generates four partial products, each of which is 32 bits wide; these must be added in at the appropriately-aligned bit positions to generate an 80-bit partial product, in logic external to the multipliers. On the next cycle another 80-bit partial product is generated, and is added to the previous 80-bit partial product at the appropriate alignment offset. Figure 7 shows the 16 32-bit partial products aligned appropriately to their binary weighting, for the entire time-sequenced multiply process. The final 128-bit product can be obtained from the addition of the four 80-bit partial products on successive clock cycles.



THE Y_i - PARTIAL 16-BIT OPERANDS Y_D, Y_C, Y_B, Y_A ARE LOADED AND MULTIPLIED BY THE ENTIRE 64-BIT X OPERAND IN FOUR STEPS TO OBTAIN A 128-BIT PRODUCT AS SHOWN IN FIGURE 7.

Figure 6. A Serial-Parallel Multiplier Architecture

Totally-Parallel 64-Bit Multiplier

A speed-oriented hardware configuration takes the approach of using whatever external logic is needed for the very fastest possible 64x64 multiply operation. Figure 7 may be applied in this case also; it shows 16 32-bit partial products. (For simplicity, will assume that the configuration described here deals strictly with unsigned integers, so that the 16 partial products are unsigned.) Since 16 'S556s are being used, then the 32-bit partial products corresponding to *all* of the combinations of the partitioned multiplier and the partitioned multiplicand are all available at the same time. Now comes the crucial aspect of the design, which involves adding all of these bits in at the appropriate binary positions!

Figure 8 shows the aligned configuration of the partial products for a 64x64 multiply operation. Each dot represents an output bit of the 'S556, shown at the topmost part of Figure 8. To generate the final product, these partial products must be "compressed." This compression can be achieved by grouping product bits in a logical manner, so that "deep and short" vectors are compressed to "shallower and longer" vectors. What is really being accomplished is the addition of several (here 3, 5 or 7) bits having the same weight into a simple binary sum; and then the adding up of all of these (overlapping) sums, which is normally much easier. This two-step summation is performed by a "Wallace-tree-adder" arrangement (r1, r2, r3) in which 7 vectors of varying lengths are compressed to 2 vectors, and these are in turn presented as 2 operands to a single carry-lookahead adder. The dots shown in the inverted pyramidal array in the middle of Figure 8 represent compressed outputs generated from the first level of dots. The lowermost array of dots represent the inputs to the adder; these are "compressed" outputs from the Wallace-Tree array.

For example, group A shown in Figure 8 consists of a 3x3 column of bits. If all of these bits were binary "ones" then the result when they were added would be 3 + (3.2) + (3.4) = 21, which is representable in 5 binary positions. This is precisely what "A1" signifies; a compression of a 3x3 block, A, to a 5-bit vector, A1. The compression can easily be achieved by using a PROM, with the 9 bits of A as address lines, and the outputs as A1. The PROM used in this example is the Monolithic Memories 63S481A 30-nsec 512x8 PROM; only five of the eight output bits are used. Designers may prefer to group the bits in a different configuration from the one suggested in Figure 8; many other arrangements are possible. For example, one may group another column of three bits and thereby reduce a 4x3 block to a 6-bit vector, using 63S3281s, which are (40-nsec 4Kx8 PROMs); this approach would give a different pattern than the one in the middle of Figure 8.

Similar compressions for Group B to B1 can be performed using 63S441/1A 1Kx4 PROMs. This configuration compresses five 2-bit vectors to a 4-bit vector, which fits the 10-bit input address and 4-bit output word of the 1Kx4 PROM.

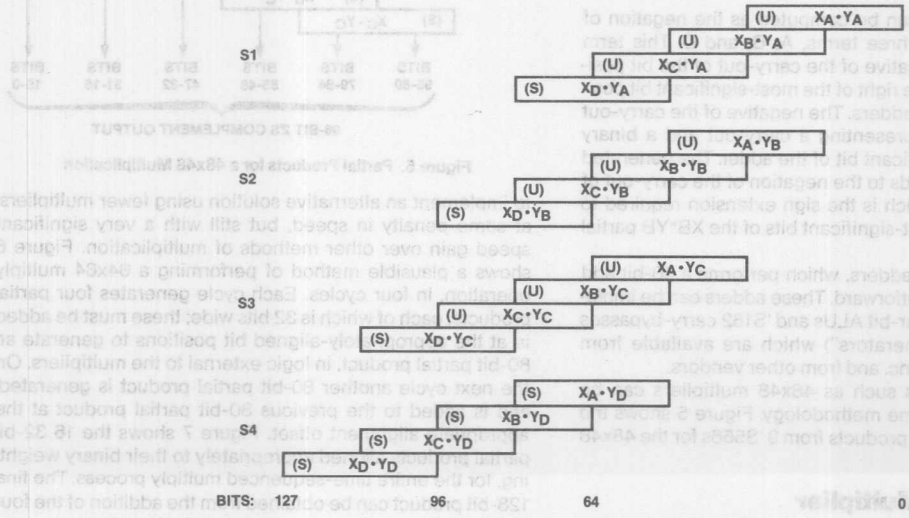


Figure 7. Partial Product Alignment for a Serial-Parallel Multiplier

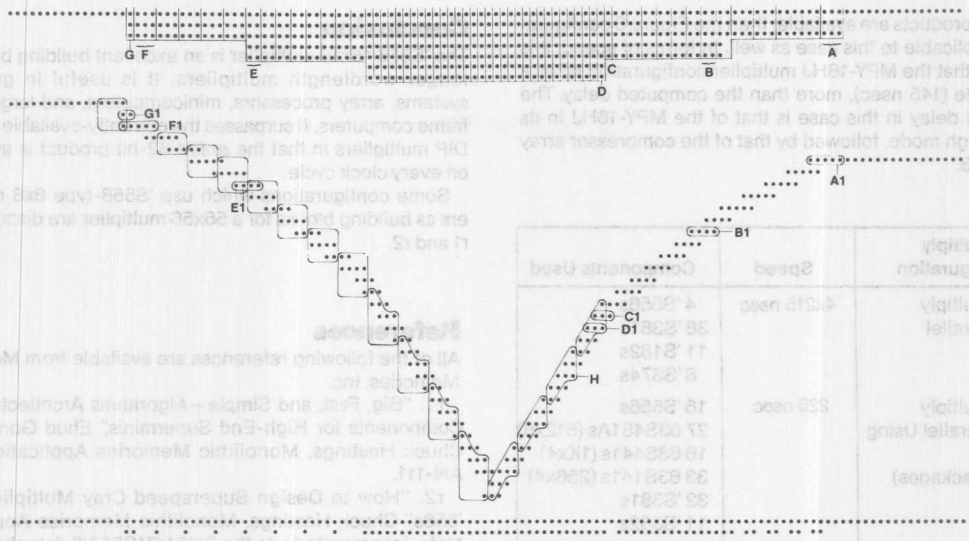


Figure 8. Partial Product/Bit Grouping for a Totally Parallel 64x64 Multiplier

Other compressions shown are C and D groups to C1 and D1 respectively. The C group is handled by compressing five 1-bit vectors to a 3-bit vector. The D group is handled by compressing seven 1-bit vectors to a 3-bit vector. The C and D groups can be compressed using 'S141/1A 256x4 PROMs. Similarly, groups E, F, G, and H are compressed to E1, F1, G1 and H1 respectively. All the above mentioned PROMs are available from Monolithic Memories.

The second level of dots has some groups of four columns. These four-column groups contain 3 bits in the least-significant bit position, and 2 bits in the remaining columns. These 9 inputs can be compressed using a 63S481A 30-nsec 512x8 PROM, to a vector 5 bits wide. For parts-counting purposes, the same 63S481A PROM type is used for all the compressions in the middle of Figure 8.

To aid users in the programming of PROMs for these and other Wallace-tree applications, or in fact any other applications exploiting PROMs as logic elements, Monolithic Memories provides *P*rogrammable *L*ogic *E*lement *AS*se*M*bler (PLEASM), a portable computer program written in FORTRAN. PLEASM provides a simple method for generating a PROM truth table. The user has only to supply equations which define the arithmetic/Boolean function needed within the PROM; PLEASM does all the drudgery of figuring out the code values which are needed in each PROM location.

Sample PLEASM source codes are shown at the end of this paper. For example, the entire 1Kx4 PROM which reduces the five 2-bit vectors to a 4-bit vector can be specified, using PLEASM, in 15 or fewer lines of code. Without PLEASM or its equivalent, the user would have had to specify the contents of 1024 PROM locations, after computing the corresponding code values for those locations.

Performance Comparisons

The bottom line for any hardware-architecture analysis is how fast the system runs, and what it costs in circuit-board real estate and dollars. With this understanding, a performance table is derived, based on three configurations.

The first is the configuration of Figure 7, using 4 'S556 multipliers; the entire multiplication takes four clock cycles. In addition to the multiplier ICs, a 64-bit adder is needed for the four partial products, which effectively furnishes a 80-bit partial product on every cycle. A 64-bit adder can be used to do the addition, since the least-significant partial-product bits are available directly. The 80-bit partial product has to be shifted 16 bits and then added to the second 80-bit partial product, which implies a need for a 64-bit register and an 80-bit ALU, which together serve as an accumulator.

The second configuration is the totally-parallel design using 16 'S556 multipliers plus PROMs and ALUs, shown in Figure 8.

The third configuration uses TRW-MPY16H-class 64-pin 16x16 multipliers. The entire 32-bit product of an MPY-16H is available on two successive clock cycles, as the product lines are shared with the incoming data. An additional 145 nsec is added to the MPY-16H time to allow for the necessary clocking and multiplexing steps to occur: effectively, the operands cannot be pipelined at one clock cycle as may be done in the 'S556 architecture. Even if the pin-compatible Am29516 multiplier is used, a cycle is still wasted, as two cycles are needed to clock the entire multiplier.

There is one way around this problem, when using the Am29516 multiplier; twice as many multipliers are used, and a pair of adjacent multipliers receive the same input operands. One multiplier of the pair then outputs the least-significant half of the product, and the other multiplier of the pair outputs the most-significant half of the product; thus, the two paired Am29516 64-pin DIPs are functionally a quasi-equivalent of the 84-pin 'S556, albeit they require many times the circuit board area.

The analysis in the Table 1 assumes the use of 16 MPY-16HJ multipliers. 16 16-bit registers are needed to hold the 16 halves of the various different partial products. After the

tion is applicable to this case as well. In terms of speed, it is assumed that the MPY-16HJ multiplier configuration takes a clock cycle (145 nsec), more than the computed delay. The computed delay in this case is that of the MPY-16HJ in its feedthrough mode, followed by that of the compressor array of Figure 8.

Multiply Configuration	Speed	Components Used
64x64 Multiply Serial-Parallel	4x215 nsec	4 'S556s 36 'S381s 11 'S182s 8 'S374s
64x64 Multiply Totally-Parallel Using 'S556s (84-pin packages)	226 nsec	16 'S556s 27 63S481As (512x8) 16 63S441s (1Kx4) 33 63S141s (256x4) 32 'S381s 11 'S382s
64x64 Multiply Clocked Parallel Using MPY-16HJ (64-pin packages)	481 nsec	16 MPY-16HJ 32 'S374s 27 63S481s (512x8) 16 63S441s (1Kx4) 33 63S141s (256x4) 32 'S381s 11 'S382s

Table 1. Performance Comparisons

Conclusion

The 'S556 16x16 multiplier is an excellent building block for longer-wordlength multipliers. It is useful in graphics systems, array processors, minicomputers, and large mainframe computers. It surpasses the currently-available 64-pin-DIP multipliers in that the *entire* 32-bit product is available on every clock cycle.

Some configurations which use 'S558-type 8x8 multipliers as building blocks for a 56x56 multiplier are discussed in r1 and r2.

References

All of the following references are available from Monolithic Memories, Inc.

r1. "Big, Fast, and Simple—Algorithms Architecture, and Components for High-End Superminis," Ehud Gordon and Chuck Hastings, Monolithic Memories Application Note AN-111.

r2. "How to Design Superspeed Cray Multipliers with '558s," Chuck Hastings, Monolithic Memories Application Note, incorporated into the SN54/74S557/8 data sheet.

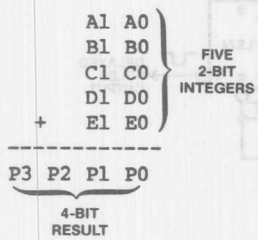
r3. "Real-Time Processing Gains Ground with Fast Digital Multiplier," Shlomo Waser, *Electronics*, 9/29/77.

r4. "State-of-the-Art in High Speed Arithmetic Integrated Circuits," Shlomo Waser, *Computer Design*, 6/1978.

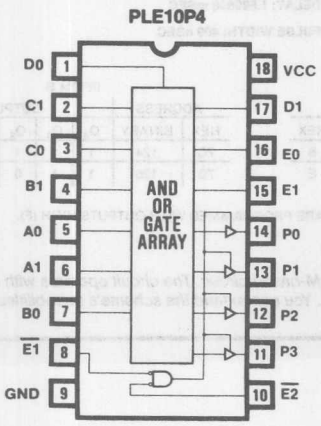
r5. "Doing Your Own Thing in High-Speed Arithmetic," Chuck Hastings, *Conference Proceedings of the 6th West Coast Computer Faire*, pages 492-510, 4/5/81. Also Monolithic Memories Conference Proceedings reprint CP-102.

Appendix—Sample PLEASM Source Listing—To Reduce Group B In Figure 8

```
PLE10P4                                          PLE DESIGN SPECIFICATION
P5020                                              VINCENT COLI 08/22/83
FIVE 2-BIT INTEGER ROW PARTIAL PRODUCTS ADDER
MMI SANTA CLARA, CALIFORNIA
.ADD A0 A1 B0 B1 C0 C1 D0 D1 E0 E1
.DAT P0 P1 P2 P3


P3,P2,P1,P0 = A1,A0 .+. B1,B0 .+. C1,C0 .+. D1,D0 .+. E1,E0  ; P = A+B+C+D+E

FUNCTION TABLE

A1 A0 B1 B0 C1 C0 D1 D0 E1 E0 P3 P2 P1 P0

;AA   BB   CC   DD   EE    PPPP     COMMENTS
;10   10   10   10   10    3210     A + B + C + D + E = P
-------------------------------------------------------------
 LL   LL   LL   LL   LL    LLLL     0 + 0 + 0 + 0 + 0 =  0
 LH   LH   LH   LH   LH    LHLH     1 + 1 + 1 + 1 + 1 =  5
 HL   HL   HL   HL   HL    HLHL     2 + 2 + 2 + 2 + 2 = 10
 HH   HH   HH   HH   HH    HHHH     3 + 3 + 3 + 3 + 3 = 15
-------------------------------------------------------------


DESCRIPTION

THIS PLE10P4 PERFORMS PARTIAL PRODUCTS REDUCTION FOR WALLACE TREE
COMPRESSION.  FIVE ROWS OF 2-BIT NUMBERS (A1-A0, B1-B0, C1-C0,
D1-D0, AND E1-E0) ARE NUMERICALLY SUMMED TO PRODUCE A 4-BIT RESULT
(P3-P0).
```



FIVE 2-BIT INTEGER ROW PARTIAL PRODUCTS ADDER



Northcon/83

Electronics Show & Convention
May 10-12, 1983
Portland, Oregon

Cascade Chapter, ERA
Portland and Seattle Sections IEEE
Portland and Seattle Chapters, NWPCA

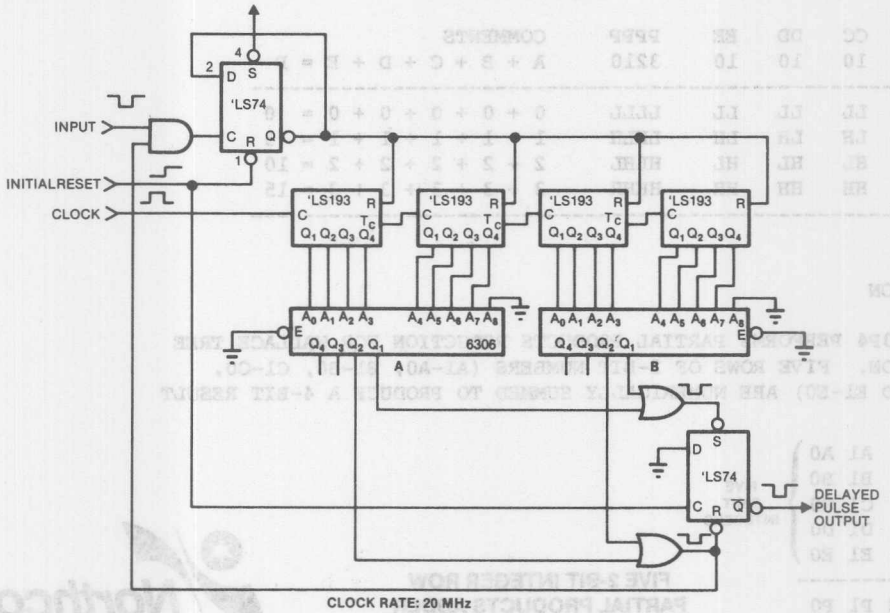
5

PROMs yield delayed pulses

Rick Wegner
Storage Tech Corp, Louisville, CO

If you need a highly accurate, delayed pulse with adjustable width, the circuit shown in the **figure** will do the job. The circuit can operate at repetition rates as high as 20 MHz, a limitation set by the bipolar PROMs' access time (in this case, 50 to 60 nsec) and the counters' maximum clock rate (32 MHz for the example's 74LS193). To generate a delayed pulse from the original input pulse, program the PROMs to yield a logic Low upon reaching the desired delay count and a logic High at the end of the delayed pulse's period.

The delay count equals the desired delay divided by the clock period. In the schematic shown, you need an OR gate because PROM A goes through several intermediate counts before both counters attain the final delay count. To set up your desired pulse width,



(a)

CLOCK RATE: 20 MHz
DESIRED DELAY: 1.599650 mSEC
DESIRED PULSE WIDTH: 400 nSEC

PROM A

ADDRESS		OUTPUT				
HEX	BINARY	Q_4	Q_3	Q_2	Q_1	HEX
01	1	1	1	0	1	B
F9	249	1	1	1	0	E

PROM B

ADDRESS		OUTPUT				
HEX	BINARY	Q_4	Q_3	Q_2	Q_1	HEX
7C	124	1	1	1	0	E
7D	125	1	1	0	1	B

ALL OTHER ADDRESSES ARE PROGRAMMED WITH OUTPUTS HIGH (F).

(b)

Obtain accurate pulse delays and widths *with this PROM-based circuit. The circuit operates with clock rates as high as 20 MHz and allows you to program very precise pulse parameters. You can extend the scheme's capabilities by adding more counters and additional PROMs.*

EDN FEBRUARY 23, 1984

program the second output from the PROMs (Q_2) to generate another pulse that represents the added delay (equal to the desired delay plus the pulse width).

These two pulses then serve to set and reset a flip flop that generates a delayed pulse with the programmed width. You can obtain additional pulses by using the PROMs' other two outputs. The feedback resets the circuit so that the next input pulse can start the delay counting again. A specific program example is shown in (**b**).

The design has several modification possibilities. If you need a longer delay (without sacrificing accuracy), you can add more counters and PROMs. Moreover, an 8-output PROM allows the generation of more delayed pulses. If you need smaller pulse widths or more accurate delays, you can disconnect PROM A's A_0 address line and shift lines A_1 through A_8 one line to the left in the schematic. This action allows the determination of a new set of delay counts, effectively doubling the input clock-rate capability.

What are the limitations of this circuit? First, because all address inputs must change simultaneously, the circuit demands synchronous counters. Second, you shouldn't use large-capacity EPROMs, because their increased access time reduces the maximum clock rate, thus reducing the accuracy of both the delay and the pulse width. **EDN**

EDN FEBRUARY 23, 1984